THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Subrata Dasgupta

TITLE OF THESIS: Parallelism in Microprogramming Systems

DEGREE FOR WHICH THESIS WAS PRESENTED: Ph. D.

YEAR THIS YEAR GRANTED: 1976

THE UNIVERSITY OF ALBERTA

PARALLELISM IN MICROPROGRAMMING SYSTEMS

by

© SUBRATA DASGUPTA

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1976

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled "PARALLELISM IN MICRO-PROGRAMMING SYSTEMS" submitted by Subrata Dasgupta in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

Parallelism in microprogramming systems is investigated here with respect to the following problems:

(a) The identification of parallel micro-operations in straight-line microprograms. Earlier solutions to this problem include algorithms which, while fairly general, do not guarantee optimal output; there are also several other algorithms, which attempt to optimize the output but are restricted in their applicability. The analysis of straight-line microprograms is extended in this thesis and a new, general, optimizing algorithm is presented.

(b) Identification of parallel micro-operations in loop-free microprograms. This is the problem of "global" parallelism (in contrast to that of "local" parallelism referred to in (a) above) and its analysis here within a graph-theoretic framework leads to a method of detecting "globally parallel" micro-operations. Global analysis may - though not necessarily - produce more optimal microcode than that produced by local analysis alone. Thus, it becomes an important strategy in designing architectures, when executional time efficiency is the main objective.

(c) Since one cannot guarantee that mechanical procedures will produce optimal microcode in an arbitrary microprogram, it seems desirable that a micro-software system should give the microprogrammer, the choice as to

whether optimization is to be performed mechanically or by the programmer. This consideration gives rise to the problem of developing language constructs for expressing horizontal (i.e. "parallel") microprograms explicitly. A solution to this problem is the third major result of this study: language constructs are proposed and their semantic features discussed. These constructs not only allow the expression of micro-parallelism, but also enable microprogram verification rules to be established analogous to rules discovered for "higher level" program statements.

(d) Potential parallelism is defined in this thesis as the parallelism embedded in the (writable) control memory (micro-) word organization. The last of the problems considered here is the analysis of potential parallelism with respect to (i) its maximization using as a basis, the assignment of micro-operations to clock-cycle phases; and (ii) its application in the determination of the smallest minimally-encoded control memory word. Previous studies of the so-called "control memory minimization problem" were concerned with read-only memories. These results are extended here to the case of writable control memories.

ACKNOWLEDGEMENTS

I would like first of all, to thank my supervisor John Tartar for his help, guidance and constant encouragement during the preparation of this thesis. I am also indebted to Dr. Len Schubert for his careful reading and critical comments on portions of this work. Dr. Barry Mailloux had very kindly read an early version of one of the chapters. I am grateful for his comments and suggestions.

I have also had the privilege of thoroughly enjoyable and fruitful discussions with Jay Weinkam and David Kirkpatrick, both of Simon Fraser University, Burnaby.

My greatest intellectual debt however, is to all those authors whose writings and ideas have so vastly influenced my philosophical views on computers and computation, and indeed on all the 'sciences of the artificial'.

I am deeply grateful to the National Research Council of Canada for their timely award of scholarships in 1974 and 1975.

Finally, a very personal note of gratitude to the 'other' Dasgupta's in my life. It is almost impossible to express how much I owe to their support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

Figure		Page

CHAPTER I

INTRODUCTION

1.1 Review

This thesis presents the results of a study of parallelism in microprogramming systems. As such it is intended as a contribution, not only to the steadily growing catalogue of microprogram optimization strategies, but also to our understanding of the nature of parallel processing systems.

The design and implementation of microprogrammed control units in fact, represents one of the earliest developments in parallel processing. For example, Wilkes' original design, and many of the initial extensions of this model (discussed by Husson [38]), fall within the category of what we now refer to as "horizontal" microprogramming. In such systems several primitive operations are executed within a basic machine cycle.

There were however, practically no attempts to analyse or develop models of, parallelism at the microprogramming level until the present decade. This can be contrasted to the extensive analysis of multiprocessing and other "higher level" concurrency phenomena which have continuously emerged since the early 1960's [6,10,12,37,53].

The very recent surge of interest in optimization, parallelism, and other "formal" aspects of microprogramming

stems from a number of reasons, the most significant of these being the emergence of the writable control memory as a technologically viable storage medium. A wide variety of machines are now commercially available [75,78,79,80], which permit users to define their own architecture through "dynamic" microprogramming, and it is easy to realize how the availability of this technique has - in theory at least - enlarged the scope of microprogramming far beyond the original objectives established by Wilkes.

As a result, extensive experimentation is currently in progress on such applications as the implementation of high-level language architectures [9,14,16,33,60,73], operating system "environments" [64,74], and "universal" host machines for emulation [24,49]. It seems likely that such applications will involve larger, and far more complex microprograms than are required for realizing simple machine language instructions. The latter of course, has been the traditional role of microprogramming.

A second consequence of writable control stores is that microprogramming is being examined more as a programming activity. As a result, several high level languages have been proposed or implemented with the primary objective of enhancing the ease of writing microprograms [15,22,26,34,47,54,56]. Numerous authors have pointed out however, that the usefulness of these languages

will depend heavily, on how efficient the object microcode is, as compared to conventionally produced microcode. These observations have thus provided the general impetus to the analysis of microprograms with a view to optimization, an area of research largely initiated by Kleir and Ramamoorthy [40].

The term "optimization" as used by these authors, refers essentially to strategies for deleting redundant micro-operations within a sequence of such operations. The analogy with program optimization is obvious, and in fact, Kleir and Ramamoorthy adopted many of the ideas originated by Allen [4,5] for machine code optimization.

From the viewpoint of <u>effectiveness</u> however, there is a small but vital distinction between program and microprogram optimizations. For, in the former case, elimination of a single instruction is "useful" in that it reduces program execution time (by the amount required to fetch and execute the deleted instruction). The deletion of a single <u>micro-operation</u> on the other hand, may not necessarily reduce microprogram execution time, since the "useful" unit of activity in this case is the <u>microinstruction</u>, which may contain several micro-operations. Deletion of a micro-operation thus, becomes useful only if the result of the deletion is the elimination of a microinstruction also. This will certainly happen in the case of vertical microprogramming systems,

but not necessarily so in horizontal schemes.

Thus in addition to techniques for optimizing microprograms in the above sense, strategies are required for compacting the microcode into as small a set of micro-instructions as possible; in other words, producing optimal or near-optimal horizontal microprograms.

The class of techniques for achieving this objective is loosely termed horizontal optimization. A major part of the present thesis is addressed to this problem, which can be stated more precisely as follows:

Let A be an algorithm to be implemented as a microprogram. Then A can be realized by a sequence of micro-operations say S, such that the sequential execution of S produces the desired result. I shall term such a micro-operation sequence, a canonical microprogram. The problem of horizontal optimization is, to determine for a given canonical microprogram (a) a partition of the micro-operations contained in it such that the micro-operations in each partition block can be executed in parallel (in some well defined sense that will be specified later), and the number of blocks for the given canonical microprogram is minimum; and (b) an ordering of the partition blocks such that the execution of the ordered set of blocks produces the same result as would be produced by the execution of the canonical microprogram. Fig. 1.1 schematizes this particular aspect of optimization.

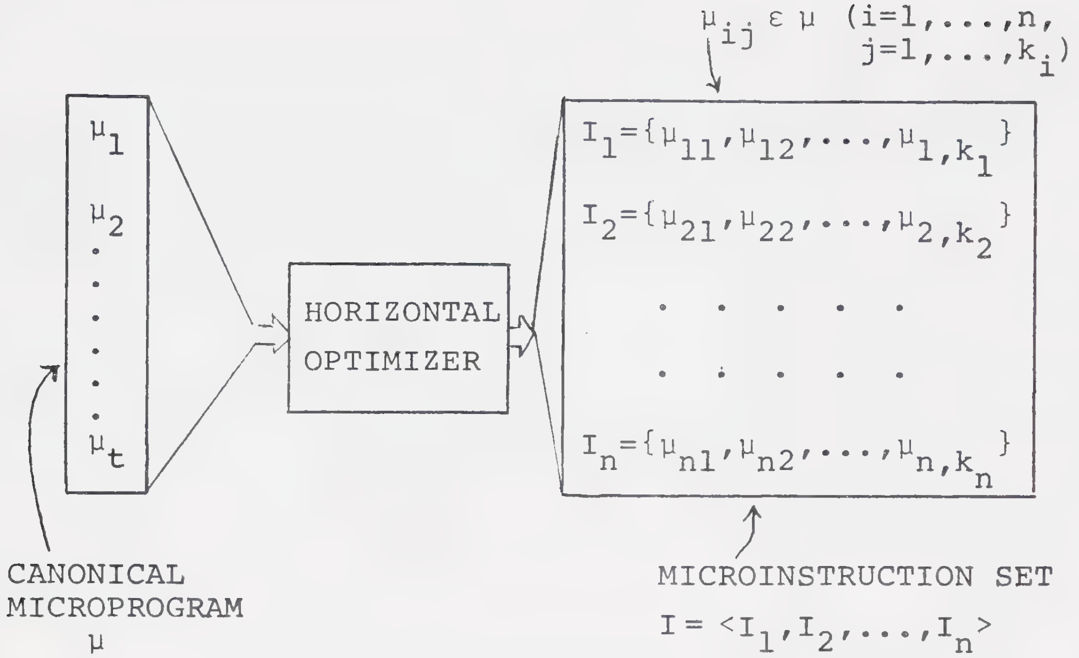

Fig. 1.1

General Scheme for Horizontal Optimization

WORDS	MICRO-OPERATIONS
1	μ_1, μ_2, μ_3, μ_4, μ_5, μ_6
2	μ_3, μ_7, μ_8, μ_9
3	μ_1, μ_2, μ_8, μ_9, μ_{10}
4	μ_4, μ_8, μ_{11}
5	μ_6, μ_8

Fig. 1.2

Sets of Micro-operations to be placed in Each Word: An Example

In order to preserve all the parallelism specified in these blocks, the <u>microinstruction word</u> ("microword") <u>organization</u> must allow each block I_j, to be placed in a word of control memory; otherwise, a part of the parallelism will be lost. For example, the block I_1 in Fig. 1.1 contains micro-operations $\mu_{11}, \mu_{12}, \ldots, \mu_{1,n_1}$. For maximum efficiency, the control memory word must be so organized as to allow these micro-operations to be specified in a single word.

I shall use the term <u>potential parallelism</u> to denote the parallelism implicit in a given microword organization.

In the design of read-only control memories (ROM's), potential parallelism is only of marginal interest. For, in this case, the precise nature of the microprograms defining the machine's instruction set is known <u>a priori</u>, and the microword organization and timing behaviour can be so determined as to maximize the <u>average actual parallelism</u> per microinstruction.

The situation for machines with writable control memories (WCM's) is however quite different: in this case, the nature of the user microprograms will be unknown to the designer. Thus, if a horizontal WCM word organization is to be used, clearly <u>one</u> of the desirable performance objectives is to enhance the microword <u>potential parallelism</u> as far as possible.

An examination of the precise nature of potential parallelism constitute a second major focus of investigation in this thesis.

The concept of potential parallelism is also useful in the context of the <u>control memory minimization problem</u>. Stated succinctly, this refers to the problem of minimizing the <u>word length</u> of control memories. Earlier, formal investigations in this area [20,32,62], were focussed principally on the minimization of read-only memories; that is, given <u>a priori</u> knowledge that specific micro-operations are to be executed in parallel, to construct a <u>minimally encoded</u> ROM word of minimal length [20,58], such that all the parallelism could be accommodated and there were no conflicts otherwise. As a specific example, Fig. 1.2 shows sets of parallel micro-operations. The problem is to determine a minimum-length, minimally-encoded word so as to permit each of the sets to be executed in parallel.

The concept of potential parallelism is applied in the present work, to extend the results of Das <u>et al</u> [19] on ROM minimization to the problem of minimizing <u>writable control memory</u> word lengths.

1.2 Defining Parallelism

In very general terms, two processes or "tasks" T_a and T_b are said to be executable in parallel if, given

a task stream containing T_a and T_b, the two tasks are mutually independent according to some criteria; if the latter are satisfied, the tasks can be executed "at the same time".

The exact nature and complexity of the criteria are determined by several factors. Specifically:

(a) The nature of the tasks;

(b) The structure of the task stream, e.g., whether the stream contains conditional branches or not;

(c) The nature of the "processors" which are to execute the tasks; and finally

(d) The quantum of time used to determine simultaneity of execution.

Consider for example, the situation where we have two identical processors sharing a main memory; we want to know under what conditions two tasks T_a, T_b, originally scheduled for sequential execution, can be initiated in parallel. Necessary and sufficient conditions were first obtained by Bernstein [10] and may be summarized by the relation

$$(SC_a \cap SK_b = \phi) \wedge (SK_a \cap SC_b = \phi) \wedge (SK_a \cap SK_b = \phi) \qquad (1.1)$$

where SC_a, SK_a (SC_b, SK_b) denote respectively, the sets of memory elements used as the data source and data sink by T_a (T_b), and ϕ denotes the empty set. These are the so-called data independency conditions.

Implicit in conditions (1.1) are the assumptions that (i) there are available two (or more) processors each of which can execute T_a and T_b; and (ii) T_a and T_b are primitive tasks for the level of processing being considered.

If T_a and T_b are non-primitive tasks, i.e., if they can be further decomposed into smaller but still meaningful tasks at the level of processing being considered, there (1.1) will not define necessary conditions. For instance, in a multiprogramming environment, there may be several, logically distinct processes executing concurrently, yet operating on a shared variable. As far as the processes are concerned, (1.1) is violated (because of the shared variable). However, by placing operations on the shared variable within critical regions, these particular operations are made mutually exclusive over time [12,13]. Yet the overall processes satisfy the intuitive notion of parallelism.

As will be seen later, parallelism at the micro-programming level involves both simultaneous and non-simultaneous processes. This is a consequence of the timing characteristics of the control unit, and the fact that by convention, the meaningful unit of activity is the microword. Stated simply micro-parallelism is the phenomenon of potential or actual activation of multiple micro-operations from a single microword. The present

dissertation is then concerned with the development, refinement, and application of this simple notion.

1.3 Organization of the Thesis

Chapter II surveys some of the earlier researches on micro-parallelism. Since much of this work was concerned with the automatic detection of parallel micro-operations in branch-free microcode, the survey is largely dominated by this topic. To provide a framework for the discussion, a model of the architecture of a "micro-programmable machine" is proposed in the earlier part of this chapter.

Chapter III develops the notion of potential parallelism; procedures for enhancing the potential parallelism in microwords, and minimizing their word lengths are presented.

In Chapter IV, a new, general, optimizing algorithm for detecting parallelism in "straight line" microprograms is presented. The proposed algorithm compares rather favourably with earlier efforts; for while the latter include at least one algorithm that is quite general in its applicability - that due to Jackson and Dasgupta [39] - it does not attempt to optimize. On the other hand, the Yau-Schoewe-Tsuchiya algorithm [77] produces

optimal output but is limited in application to monophase microprograms.

Extending the detection of parallelism to branch-containing microprograms becomes important when efficiency of (microprogram) execution is of prime consideration. The problem of what I call "global" parallelism (in contrast to the "local" parallelism in straight line microprograms) is analysed using graph-theoretic concepts in Chapter V, and a system of algorithms is developed for detecting both local and global parallelism in "loop-free" microprograms.

It must be remembered that the whole idea of horizontal optimization stems from the premise that microprograms will be written in sequential form in some high level language, and that optimization will be performed by the compiler. I feel however, that the microprogrammer should in fact, be given a choice as to whether optimization is to be done mechanically or manually; this seems rather important given the fact that there are limits to the extent of mechanical optimization that is feasible. Thus highly used segments of microcode can be optimized by the programmer, leaving less frequently used segments to the compiler.

The above considerations lead to the interesting problem of developing language constructs for horizontal microprogramming.

This problem forms the subject matter of Chapter VI. I will argue that it is not merely sufficient for a particular set of constructs to express microparallelism, they should also allow microprogram verification rules to be established in the tradition of similar rules discovered by Hoare for higher level programming statements [35,36]. A specific set of constructs are proposed in Chapter VI, and their features discussed. Verification rules for these constructs are also determined.

CHAPTER II

MICROPARALLELISM: A FRAMEWORK AND SURVEY

2.1 Architecture of the Microprogrammable Processor

A microprogrammable processor (MP) is simply the processor "as seen by" the (microprogrammed or microprogrammable) control unit. Given below, is a model of the MP which can serve as a framework for much of the discussion that follows.

An MP is characterized by (i) a set of resources $R = \bar{M} \cup \bar{O} \cup \bar{P}$ where $\bar{M}$ is a set of memory elements, $\bar{O}$ a set of operational units, and $\bar{P}$ a set of data-paths; and (ii) a set of feasible events $\bar{E}$. Examples of elements from the sets of (i) are respectively, registers, the arithmetic-logic unit (ALU) and a path between a register output and an ALU input.

An event $E \varepsilon \bar{E}$ can be one or a combination of the following:

(a) a simple flow of information along a path $P \varepsilon \bar{P}$;

(b) a registration of information in a memory element $M \varepsilon \bar{M}$; or

(c) the activation of a unit $O \varepsilon \bar{O}$ thereby causing O to perform a computation. In such a case it is assumed that O simply extracts the argument on its ports, computes the desired function, and presents the result on its output port.

For example, let M be a memory element, 0 a shift unit, and P a path from M to 0's input. Then two possible events may be described symbolically by:

$$0\text{INPUT} \xleftarrow{P} M; \tag{2.1}$$

$$0\text{OUTPUT} \xleftarrow{0} \underline{\text{shiftleft}}\ (0\text{INPUT}); \tag{2.2}$$

The first event causes the contents of M to be transferred (along P) to 0's input; the second event causes a "shift left" operation to be performed by 0 on its input argument.

In the MP, an event is caused by a <u>control</u> <u>signal</u> originating in a read-only memory (ROM) or a writable control memory (WCM); each such signal is termed a <u>micro-operation</u> (MO). Let the set of all MO's be denoted by μ^*. It is assumed that μ^* is a finite set, and that there exists a one to one correspondence between elements of μ^* and elements of E. Because of this correspondence, the term "micro-operation" can be used without ambiguity, to denote both the control signal and the event invoked by the signal.

The control memory is considered to be a linear sequence of words (<u>microwords</u>). Each microword in turn is composed of a set of subwords or <u>fields</u>. The precise organization of the fields is not of importance for the present. For our purposes, only those fields are of

interest which are directly responsible for the execution of the micro-operations. More precisely, each such field F_i is associated with, or is an encoding of, a specific subset of MO's from μ^*:

$$F_i = \{\mu_{i1}, \mu_{i2}, \ldots, \mu_{i,k_i}\} \tag{2.3}$$

such that at any given time, one and only one of these MO's can be executed. Thus MO's encoded in the same field are mutually exclusive over time.

The execution of MO's is controlled by a <u>machine cycle</u> C, characterized by a set of phases $\Pi_1, \Pi_2, \ldots, \Pi_k$ (Fig. 2.1) such that, each MO is executed in a specific phase or sequence of phases of C; or (less frequently), the MO's execution time spans several cycles. The present model assumes that all MO's are synchronous. The phases of the machine cycle may overlap, as shown in Fig. 2.1.

A <u>microinstruction</u> I is a (microprogrammer) specified set of MO's executed (or to be executed) from a single microword. The relation between microword and microinstruction is analogous to that of the class of instructions of a particular format, and an instruction <u>of</u> that format at the machine instruction level. One must note however that in the case of ROM's, the microword has no separate identity of its own since each microword in the ROM is in effect, a distinct microinstruction.

As stated above, MO's are assumed to be activated in one or more phases of the machine cycle C, or over

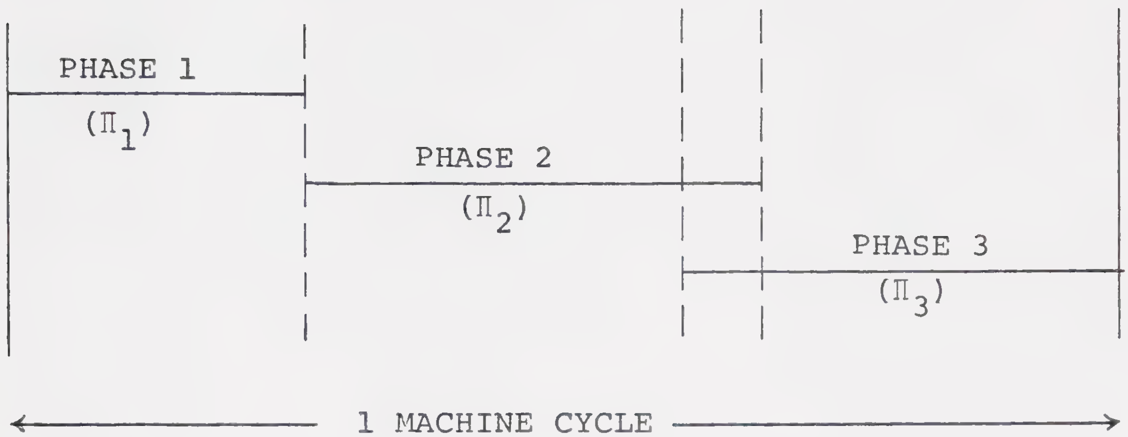

Fig. 2.1

A Polyphase Machine Cycle

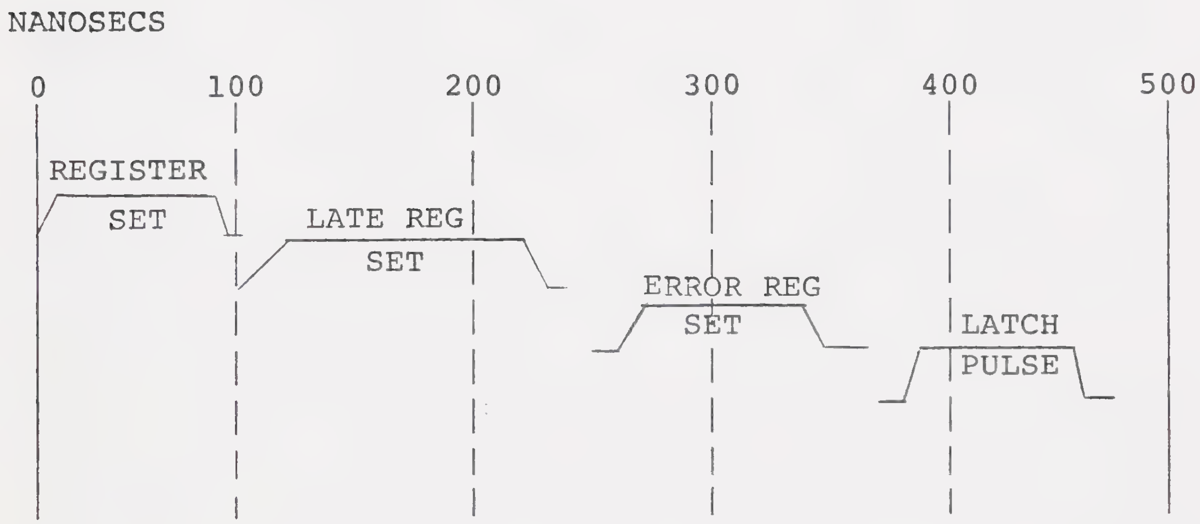

Fig. 2.2

A Non-Overlap Polyphase Machine Cycle

several such cycles. A further aspect of timing is the assumption that the execution of a <u>microinstruction</u> requires one or more machine cycles; that is, if t_c denotes the duration of a machine cycle, then a microinstruction I_j is executed in time $N_j t_c$ for some integer $N_j \geq 1$ depending on I_j. In the usual case $N_j = 1$, i.e., the microinstruction cycle time is the same as the machine cycle. The most common situation under which $N_j > 1$ is if I_j includes a main memory read, or write operation. Fig. 2.2 provides an instance of a machine cycle containing a number of non-overlapping phases; the class of operations associated with each phase is also indicated.

A <u>microprogram</u> is any sequence of microinstructions the execution of which causes a machine instruction from main memory to be partially or completely interpreted.

This completes the description of the MP. Further elaboration or refinements of this architecture will follow in relevant sections of the thesis. I shall complete this section however, by introducing a useful notation for denoting MO's. The particular representation used here, has evolved from notations proposed originally by Kleir and Ramamoorthy [40], developed further by Sitton [63] and later modified by Jackson and Dasgupta [39].

A micro-operation will be denoted by the 5-tuple

$$\mu = < OP, SC, SK, U, V > \tag{2.4}$$

where

'OP' designates a primitive <u>operation</u>, e.g., ADD, SHIFT, GATE;

'SC', 'SK' denote the data <u>source</u> and <u>sink</u> sets respectively for 'OP';

'U' denotes the set of <u>operational</u> units and/or paths required to execute μ. U will be simply called μ's <u>unit</u>; and

'V' is a symbol representing the phase(s) of the machine cycle, or the number of such cycles in which μ is executed. V is called the <u>time-validity</u> of μ.

If the MO simply involves information flow along a path, then the U field can be left unspecified as long as the path is implicitly defined by the SC and SK fields. Finally, given the above representation, μ's <u>resources</u> are given by $R_\mu = SC \cup SK \cup U$. Some examples of MO's using the above representation are:

$$
\begin{aligned}
\mu_1 &= \langle \text{GATE}, \{\text{M1}\}, \{\text{ALU-LEFT}\}, \_\_\_\_\_, \Pi \rangle \\
\mu_2 &= \langle \text{NOT}, \{\text{ALU-LEFT}\}, \{\text{ALU-OUT}\}, \{\text{ALU}\}, \Pi_2 \rangle \\
\mu_3 &= \langle \text{GATE}, \{\text{ALU-OUT}\}, \{\text{M2}\}, \_\_\_\_\_, \Pi_3 \rangle \\
\mu_4 &= \langle \text{SHL}, \{\text{M2}\}, \{\text{M2}\}, \{\text{SHFTR}\}, \Pi_2 \rangle
\end{aligned}
\tag{2.5}
$$

2.2 Analysis of Straight Line Microprograms

At the time of writing, most of the work on detecting parallel micro-operations were concerned with straight line microprograms (SLM's). An SLM is simply, a sequence

of micro-operations.

$$S = \langle \mu_1, \mu_2, \ldots, \mu_t \rangle$$

with a single entry point (μ_1) and a single exit point (μ_t). The term is thus synonymous with "basic block" as used in the theory of program optimization [2,5]. In this section I shall review some of the known results on SLM's.

As stated in Chapter I, microprogram optimization techniques were pioneered by Kleir and Ramamoorthy [40]. The same authors attempted to formulate precisely, the conditions necessary and sufficient for microparallelism: two micro-operations μ_i, μ_j, could be placed in the same microinstruction provided that

$$(SC_i \cap SK_j = \phi) \wedge (SK_i \cap SC_j = \phi) \wedge (SK_i \cap SK_j = \phi) \wedge (U_i \cap U_j = \phi) \tag{2.6}$$

Note that (2.6) is stronger than Bernstein's condition (1.1). As pointed out earlier, Bernstein assumed the availability, at all times, of at least two processors capable of executing both tasks. Such an assumption is hardly valid for microprograms since an MO is executed by a specialized and (usually) unique unit. Hence the condition $U_i \cap U_j = \phi$ must be explicitly stated.

Kleir and Ramamoorthy also pointed out that Tomasulo's algorithm for multiple hardware units [70]

could be adopted to the analysis of microprograms. The basic idea is to examine the data flow and determine which outputs can be fanned out to memory elements in parallel, thereby eliminating temporary storage. For instance, consider the micro-operation sequence in Fig. 2.3 [55].

Since a gating operation simply transfers data between resources, the data in R_3, R_5, and R_6 are identical after μ_{14} has been executed. If however, the result of the operation $R_1 + R_2$ (in μ_{11}) could be concurrently fanned out to more than one sink unit, μ_{13} and μ_{14} would become redundant, hence deletable. The resulting sequence is shown in Fig. 2.4.

μ_{11} : $R3 \leftarrow R1 + R2;$
μ_{12} : $R4 \leftarrow R3 - R4;$
μ_{13} : $R5 \leftarrow R3;$
μ_{14} : $R6 \leftarrow R5;$
μ_{15} : $R5 \leftarrow R6 \wedge R1;$
μ_{16} : $R7 \leftarrow R3 + R1;$

Fig. 2.3 (1)

Input to Tomasulo's Algorithm

μ'_{11} : $R3, R6 \leftarrow R1 + R2$
μ'_{12} : $R4 \leftarrow R3 - R4;$
μ'_{13} : $R5 \leftarrow R6 \wedge R1;$
μ'_{14} : $R7 \leftarrow R3 + R1;$

Fig. 2.4

Output of Tomasulo's Algorithm

---

(1) The notation used here is based on Tsuchiya's SIMPL language [54,71]. Whenever convenient, I shall use this notation in conjunction with, or as an alternate to, (2.4).

Actually, since this particular strategy was proposed as a means of identifying redundant (hence deletable) MO's, it belongs more properly to "vertical" optimization. Its interest in the present context lies in its utilization of knowledge of the potential parallelism in the machine data flow.

Kleir and Ramamoorthy's condition (2.6) are however, not sufficiently general to include parallelism in polyphase systems. A more complete analysis of the problem, taking polyphase schemes into account was subsequently reported by Jackson and Dasgupta [39]. The main result of this analysis uses the following notations and concepts:

For a pair of MO's, μ_i, μ_j, the relation $\mu_i \alpha \mu_j$ is said to hold if $(SC_i \cap SK_j = \phi) \wedge (SK_i \cap SC_j = \phi)$. If in addition, the condition $(SK_i \cap SK_j = \phi)$ is satisfied, then $\mu_i \beta \mu_j$. Intuitively then, $\mu_i \beta \mu_j$ implies that μ_i, μ_j are <u>data independent</u>.

If for a pair of MO's μ_i, μ_j, the time validities V_i, V_j, are identical, or they overlap, then this is denoted by $V_i \cap V_j \neq \phi$. If V_i <u>precedes</u> V_j with respect to the reference machine cycle, then $V_i < V_j$ (or $V_j > V_i$). Furthermore, $V_i < V_j$ or $V_j < V_i$ implies $V_i \cap V_j = \phi$. Finally, if an MO μ_i, <u>precedes</u> an MO μ_j in an SLM, then $\mu_i < \mu_j$.

<u>Definition 2.1</u>

A pair of MO's μ_i, μ_j in an SLM S, satisfying

$\mu_i < \mu_j$, are disjoint, denoted $\mu_i \,\delta\, \mu_j$ if any of the following conditions are satisfied:

(i) $(V_i \cap V_j \neq \phi) \wedge (\mu_i \,\beta\, \mu_j) \wedge (U_i \cap U_j = \phi)$;

(ii) $(V_i > V_j) \wedge (\mu_i \,\beta\, \mu_j)$;

(iii) $(V_i < V_j) \wedge (\mu_i \,\alpha\, \mu_j)$.

Statement (i) of this definition simply lists the conditions for simultaneous execution of two MO's. That is, if the time validity fields intersect, and the MO's are in the same microinstruction, then there must be neither unit conflicts nor data dependencies between the MO's. The other two statements merely relax the conditions on hardware resources in order that the MO's be parallel.

Definition 2.2

A pair of MO's μ_i, μ_j in an SLM satisfying $\mu_i < \mu_j$ are conditionally disjoint, denoted $\mu_i \,\gamma\, \mu_j$ provided that

$$(V_i < V_j) \wedge \sim (\mu_i \,\alpha\, \mu_j) \quad .$$

This definition in fact states the condition under which a pair of MO's may be placed in the same microinstruction even though their resources are in conflict. For example consider the following:

$\mu_1 = \langle \text{GATE}, \{ A \}, \{ B \} , \text{———} , V_1 \rangle$

$\mu_2 = \langle \text{ADD}, \{B,C \}, \{ D \} ,\{\text{ADDER}\}, V_2 \rangle$

Notice here that $SC_2 \cap SK_1 \neq \phi$; however, if $V_1 < V_2$ then it is immaterial that the sources and sinks intersect since μ_1 will be activated (and terminated) before μ_2 begins execution even when they are placed in the same micro-instruction.

Based on these definitions, the conditions necessary and sufficient for pairwise parallelism between MO's were obtained in [39] as follows: For a pair of MO's μ_i, μ_j in an SLM satisfying $\mu_i < \mu_j$, μ_i and μ_j are parallel (denoted $\mu_i || \mu_j$) iff

$$(\mu_i \, \delta \, \mu_j) \vee (\mu_i \, \gamma \, \mu_j) \tag{2.7}$$

For a proof the reader is referred to [39]. Considering the SLM specified below, it can be seen for example, that $\mu_1 || \mu_2$, $\mu_1 || \mu_3$ $\sim(\mu_2 || \mu_3)$, $\sim(\mu_3 || \mu_4)$, and $\mu_4 || \mu_5$.

$$\begin{aligned}
\mu_1 &= \langle \text{ ADD}, && \{5,6\} && \{4\}, && \{\text{ADDER}\}, && \Pi_1 \rangle \\
\mu_2 &= \langle \text{ SHFTR}, && \{7\}, && \{7\}, && \{\text{SHIFTER}\}, && \Pi_2 \rangle \\
\mu_3 &= \langle \text{ GATE}, && \{8\}, && \{7\}, && -, && \Pi_2 \rangle \\
\mu_4 &= \langle \text{ GATE}, && \{9\}, && \{8\}, && -, && \Pi_1 \rangle \\
\mu_5 &= \langle \text{ GATE}, && \{10\}, && \{9\}, && -, && \Pi_2 \rangle
\end{aligned} \tag{2.8}$$

$$\Pi_1 < \Pi_2$$

2.3 Algorithms for Identifying Parallelism in SLM's

Procedures for identifying parallelism in SLM's have been developed in recent years by several authors, notably Ramamoorthy and Tsuchiya [54], Jackson and Dasgupta [39], Tsuchiya and Gonzales [72], and Yau *et al* [77]. These are discussed below.

In assessing these algorithms, one should keep in mind the following three principal measures of performance:

(i) the generality of the algorithm;

(ii) its optimality; and

(iii) the complexity of the algorithm.

By *generality*, I mean the extent to which the algorithm is applicable to a broad class of machine structures, timing attributes and microinstruction forms. This is necessarily a qualitative measure. For instance if an algorithm ignores the machine's timing characteristics, then clearly it is applicable to monophase systems only.

By *optimality*, I am really referring to the algorithm's optimizing *capability*, i.e., how close the output set of microinstructions is to some "minimum".

Given a (canonical) microprogram

$$S = \langle \mu_1 \mu_2 \cdots \mu_t \rangle$$

and two optimizing algorithms A_1, A_2, A_1 will be said to be "more optimal" than A_2 if A_1 when applied to S produces

$N_1(S)$ microinstructions, A_2 when applied to S produces $N_2(S)$ microinstructions and $N_1(S) < N_2(S)$.

Now, given S, equivalent microprograms may be produced by reordering (permuting) the MO's of S. By "equivalent" is meant that for all initial inputs to the microprograms, identical outputs are produced. Hence A_1 achieves greater optimality (with respect to A_2) if A_1 (implicitly or explicitly) transforms S into some equivalent microprogram S', and A_2 transforms S into some equivalent microprogram S" such that $N_1(S') < N_2(S")$.

The reader should note that I am considering reordering only, as a means of transformation; another source of transformation is to search for a sequence of MO's (say S*) from all possible sequences of MO's such that S* is computationally equivalent to S. Such transformations will be ignored here.

Finally, by complexity, I refer to the computational complexity of the algorithm. An appropriate measure of complexity in the present context is the number of pairwise comparisons of MO's performed by the algorithm, as a function of the SLM's length.

2.3.1 The Ramamoorthy-Tsuchiya Algorithm [54]

This algorithm (henceforth denoted as the RT algorithm) is composed of four phases as follows:

Phase 1: The input SLM is scanned, and the data dependencies between MO's established. Using this information,

a <u>dependency graph</u> is constructed by the method developed by Ramamoorthy and Gonzales in [31,53]. Thus, for the sequence shown in Fig. 2.5, the dependency graph is as indicated in Fig. 2.6.

μ_{21} : ACC ← R1 ∧ M3;

μ_{22} : R4 ← R2 ∧ M3;

μ_{23} : ACC ← R4 + ACC;

μ_{24} : R3 ← R3 V ACC;

μ_{25} : R1 ← R1 ∧ M4;

μ_{26} : R2 ← R2 ∧ M4;

μ_{27} : ACC ← RO ;

<u>Fig. 2.5</u>

<u>RT Algorithm: Input Example</u>

Note that the graph is not based on the data independency relation "β", alone. For instance $\sim(\mu_{22}\ \beta\ \mu_{26})$; yet according to the dependency graph they are independent. This is because, in deriving data dependencies, the algorithm assumes a specific timing scheme (Fig. 2.7). Thus, the contents of R2 will have been gated out (in μ_{22}) before a new value is gated into R2 (in μ_{26}).

<u>Phase 2</u>: The <u>earliest</u> and <u>latest</u> possible execution times for each MO are determined using the method

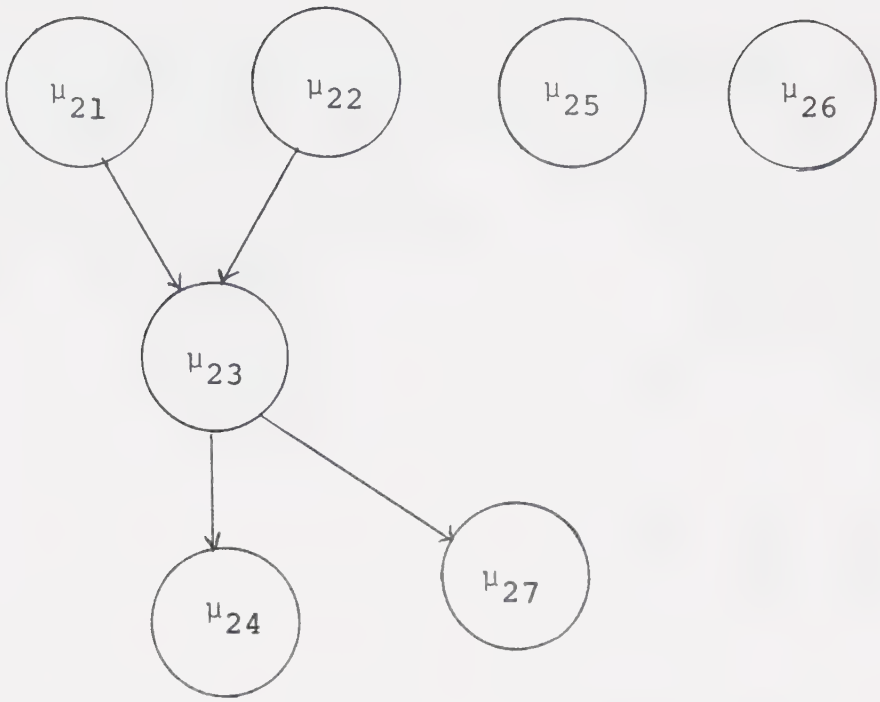

Fig. 2.6

Dependency Graph for the RT Algorithm: An Example

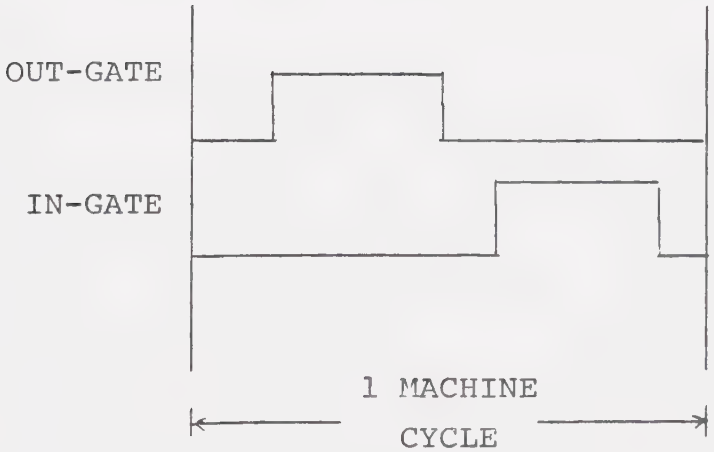

Fig. 2.7

Timing Scheme for the RT Algorithm

described in [53]. For the particular example of Fig.2.5, the earliest and latest times are shown in Figs. 2.8 and 2.9 respectively; here, t_1,t_2 and t_3 designate three successive time "frames".

t_1 : μ_{21}; μ_{22}; μ_{25}; μ_{26};

t_2 : μ_{23};

t_3 : μ_{24}; μ_{27};

Fig. 2.8

Earliest Times

t_1 : μ^*_{21}; μ^*_{22}

t_2 : μ^*_{23};

t_3 : μ_{26}; μ^*_{24}; μ^*_{27}; μ_{25};

Fig. 2.9

Latest Times

Phase 3: Critical MO's are defined as those MO's which occupy the same time frames in both the earliest-time and latest-time tables. In this phase, critical MO's are identified (asterisked MO's in Fig. 2.9).

Phase 4: Each set of concurrently executable MO's are assigned to a single time frame, the latter designating a single clock cycle. Critical MO's in the same time frame are first compared for unit conflicts; if conflicts exist, they are ordered so as to resolve these conflicts. For instance μ_{21},μ_{22} use the logic unit, hence they must be placed in different time frames in spite of their data independency. The critical MO's alone give rise to the time frames shown in Fig. 2.10. The non-critical MO's are then placed in the earliest possible time frame that

does not generate any resource conflicts.

t_1 : μ_{21};

t_2 : μ_{22};

t_3 : μ_{23};

t_4 : μ_{24}; μ_{27};

Fig. 2.10

Time-frames for Critical MO's

$I_1(t_1)$: μ_{21};

$I_2(t_2)$: μ_{22};

$I_3(t_3)$: μ_{23}; μ_{25};

$I_4(t_4)$: μ_{24}; μ_{27};

$I_5(t_5)$: μ_{26}.

Fig. 2.11

Final Output from the RT Algorithm

The final output produced by Phase 4 is shown in Fig. 2.11.

The RT algorithm is essentially an adaptation of Ramamoorthy and Gonzales' method for detecting parallel tasks in a multiprocessor system [53]. The reader will note that data independencies and unit conflicts are determined in separate phases; this will reduce computational time to the extent that unit conflicts between critical MO's need to be resolved only if the critical MO's are in the same time frame (are data-independent). However, conflict analysis involving non-critical MO's may not be so economical; e.g., μ_{26} though belonging to t_1 in the earliest-time table (Fig. 2.8), is eventually placed in I_5 (Fig. 2.11); this implies that μ_{26} is

compared with at least one MO from each time frame.

From the viewpoint of generality, the RT algorithm is limited to the extent that a specific assumption is made about the timing constraints (Fig. 2.7). While this assumption is valid for certain machines, far more complex polyphase timing schemes may also exist. The applicability of the algorithm to such systems is not at all evident.

The microcode produced by the RT algorithm is non-optimal. However, I shall describe below an extension of the method which attempts to produce optimal output. Finally, a worst-case analysis of the algorithm indicates that it requires $O(n^2)$ comparisons, n being the length of an input SLM. The critical phase here is Phase 1, where all the MO's may have to be compared on a pairwise basis.

2.3.2 The Jackson-Dasgupta (JD) Algorithm [39]

This algorithm is based on the results discussed in section 2.2, in particular, condition (2.7). There are essentially two phases:

Phase 1: Using (2.7), a conflict graph $G = \langle V,E \rangle$ is constructed, in which V, the set of vertices, designates MO's, and the set of edges E is constructed according to the following rules:

(i) there is an unlabelled edge from μ_i to μ_j if $\sim(\mu_i || \mu_j)$;

(ii) there is a labelled edge from μ_i to μ_j if $\mu_i \gamma \mu_j$.

In other words, if an unlabelled edge from μ_i to μ_j exists, then μ_i must precede μ_j; if a labelled edge exists, μ_i and μ_j are conditionally disjoint (Def. 2.2).

As an example, consider the SLM of Fig. 2.12 below. Assuming $\Pi_1 < \Pi_2$, the conflict graph will be as indicated in Fig. 2.13. The labelled edges are indicated by 1's.

μ_1 : $\langle$ GATE, {A} , {B} , _____ , Π_2 $\rangle$

μ_2 : $\langle$ GATE, {B} {D} , _____ , Π_1 $\rangle$

μ_3 : $\langle$ GATE, {C} {D} , _____ , Π_2 $\rangle$

μ_4 : $\langle$ ADD , {D,E}, {D} , {ALU} , Π_1 $\rangle$

μ_5 : $\langle$ AND , {F,A}, {A} , {ALU} , Π_1 $\rangle$

μ_6 : $\langle$ GATE, {A} , {B} , _____ , Π_2 $\rangle$

μ_7 : $\langle$ OR, {D,F}, {F} , {ALU} , Π_1 $\rangle$

μ_8 : $\langle$ GATE, {F} , {G} , _____ , Π_2 $\rangle$

Fig. 2.12

Input SLM to the JD Algorithm

Phase 2: From the conflict graph, sets of parallel MO's are extracted iteratively as follows:

(i) If $V = \phi$ then stop else

construct a set I of vertices from V such that the indegree of each vertex in I is zero;

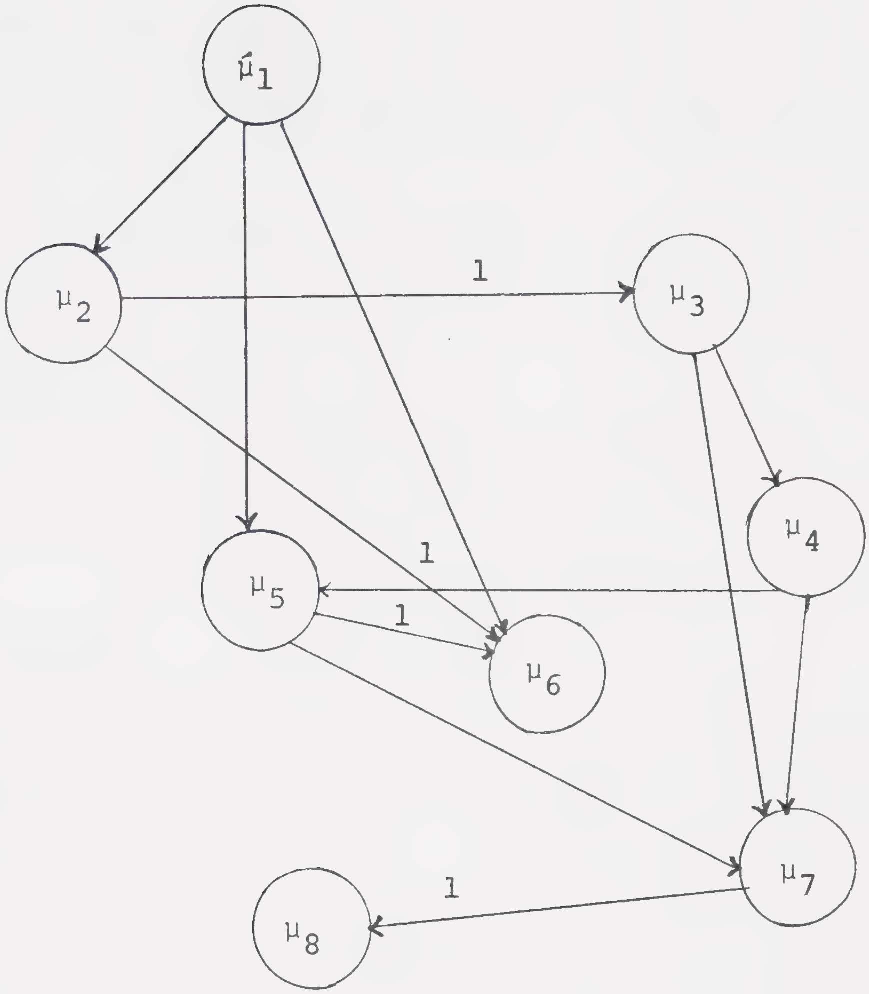

Fig. 2.13

Conflict Graph for the JD Algorithm: An Example

(ii) <u>While</u> V-I contains a vertex μ_i satisfying

(a) all edges terminating at μ_i are labelled, and

(b) all edges terminating at μ_i originate from I

<u>then</u> $I \leftarrow I \cup \{\mu_i\}$;

(iii) Output I as a set of parallel MO's;

(iv) Form a subgraph using the vertices V-I; that is

$$V \leftarrow V - I; \; E \leftarrow E \cap ((V-I) \times (V-I));$$

(v) <u>Goto</u> [i].

Intuitively, a vertex μ_i of 0 indegree implies that all MO's that must precede it have been placed in an earlier microinstruction. Thus MO's selected in step [i] of each iteration are pairwise parallel by virtue of the δ relation. MO's selected in step [ii] are conditionally disjoint to some MO in I and have no conflicts with any other MO in the graph; hence, they can also be placed in I by virtue of the γ relation. Since the graph is reduced after each iteration, the algorithm finally terminates when V becomes empty. For the example of Fig. 2.12, the output obtained is:

I_1 : $\{\mu_1\}$;
I_2 : $\{\mu_2', \mu_3\}$;
I_3 $\{\mu_4\}$;
I_4 : $\{\mu_5, \mu_6\}$;
I_5 : $\{\mu_7, \mu_8\}$;

<u>Fig. 2.14</u>

<u>Output produced by the JD Algorithm</u>

The JD algorithm is more general in its applicability than the RT algorithm since the only assumption made about the underlying machine structure is that MO's be representable unambiguously as 5-tuples (2.4). Note that timing schemes containing any number of phases are permissible, and that phases may even overlap.

Like the RT algorithm the JD algorithm is of complexity $O(n^2)$ (where $n =$ length of the SLM), since $O(n^2)$ comparisons between MO pairs are required to construct the conflict graph. Given the conflict graph however, Phase 2 of the algorithm can be rather efficiently implemented [21], by following the ideas proposed by Knuth for his topological sorting algorithm [42]. The timing of the algorithm is given by $K_1N + K_2M$ where N is the number of edges and M, the number of vertices in the conflict graph, and K_1, K_2 are constants.

Like the RT algorithm, the JD algorithm does not attempt to optimize the microcode.

2.3.3 The Tsuchiya-Gonzales (TG) Algorithm [72]

This is a refinement of the RT algorithm with the objective of producing where possible, more optimal code than is produced by the RT method.

As in the latter, the SLM is partitioned to indicate the earliest and latest execution times. However within each time frame, MO's are further partitioned by

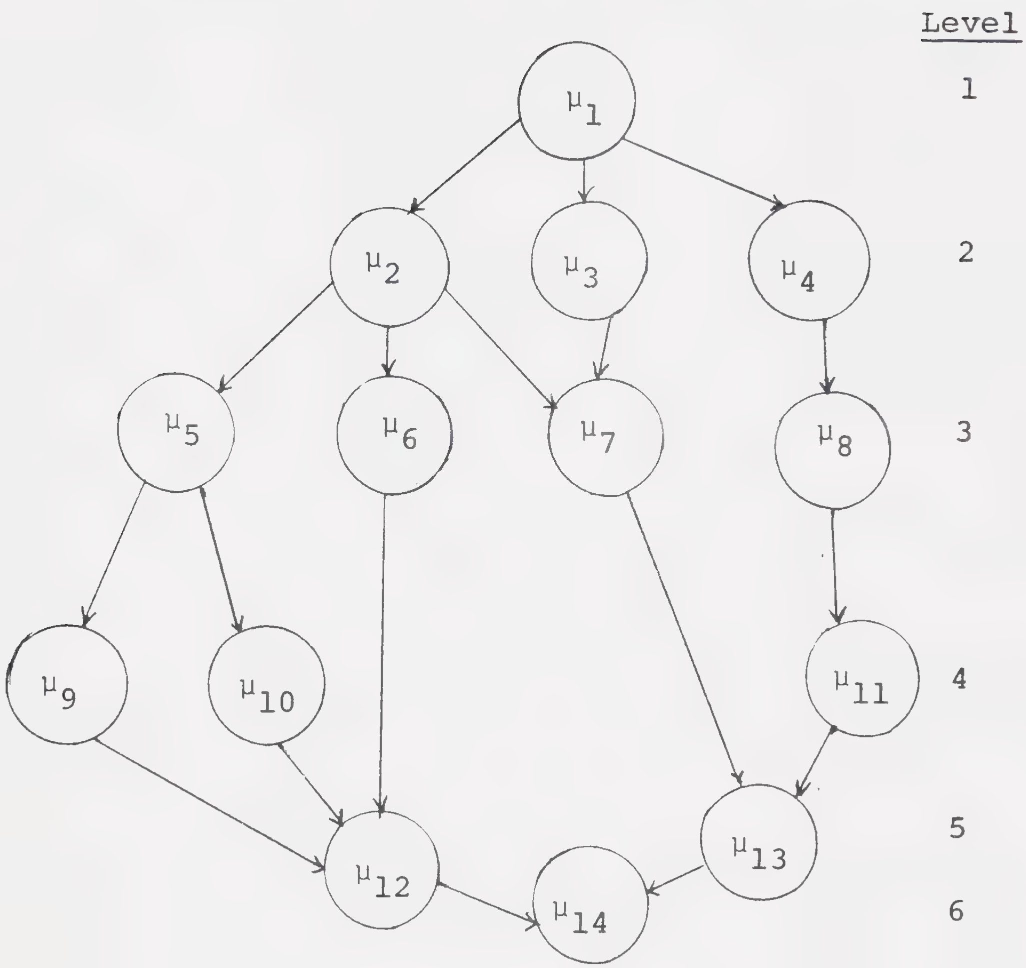

Fig. 2.15

Dependency Graph for the TG Algorithm

their resource types. The information required for this step is obtained from a <u>resource usage matrix</u> R whose rows correspond to MO's and columns to resources, and

$$R_{ij} = 1 \quad \text{if } \mu_i \text{ uses resource } j;$$
$$= 0 \quad \text{otherwise.}$$

Consider the dependency graph of Fig. 2.15: its earliest (E) and latest (L) time partitions are shown as Fig. 2.16, while the partitioning of L according to resource usage, is indicated in Fig. 2.17.

Thus, $(\mu_7, \mu_{10}, \mu_{11})_A$ simply indicates that the MO's $\mu_7, \mu_{10}, \mu_{11}$ all require resource A.

The algorithm is best understood by applying it to an example. Consider for instance, the example represented by Fig. 2.15.

<u>Step [1]</u>: L_1 is examined and is found to contain only one MO. The corresponding partition in E, viz., E_1 is then scanned, but since there are no other MO's in E_1, the microinstruction $I = \{\mu_1\}$ is constructed.

$$E = \{E_1, E_2, E_3, E_4, E_5, E_6\}$$

$E_1 = (\mu_1)$; $\qquad E_4 = (\mu_9, \mu_{10}, \mu_{11})$:

$E_2 = (\mu_2, \mu_3, \mu_4)$ $\qquad E_5 = (\mu_{12}, \mu_{13})$:

$E_3 = (\mu_5, \mu_6, \mu_7, \mu_8)$; $\qquad E_6 = (\mu_{14})$:

<u>Fig. 2.16(a)</u>

<u>Earliest time partition for the dependency graph of Fig. 2.15</u>

$L = \{L_1, L_2, L_3, L_4, L_5, L_6\}$

$L_1 = (\mu_1);$ $\quad$ $L_4 = (\mu_6, \mu_7, \mu_9, \mu_{10}, \mu_{11}):$

$L_2 = (\mu_2, \mu_4);$ $\quad$ $L_5 = (\mu_{12}, \mu_{13}):$

$L_3 = (\mu_3, \mu_5, \mu_8);$ $\quad$ $L_6 = (\mu_{14}):$

Fig. 2.16(b)

Latest time partition for the dependency graph of Fig. 2.15

$[(\mu_1)_{RO}];$

$[(\mu_2)_{B,R1}, \quad (\mu_4)_{L,R4}];$

$[(\mu_3)_{B,R2}, \quad (\mu_5)_{R5}, \quad (\mu_5, \mu_8)_L, \quad (\mu_8)_{R7}];$

$[(\mu_6, \mu_9)_S, \quad (\mu_6, \mu_{10})_{R7}, \quad (\mu_7)_{R6}, \quad (\mu_9)_{R4},$

$(\mu_7, \mu_{10}, \mu_{11})_A, \quad (\mu_{11})_{R8}];$

$[(\mu_{12})_{S,R7}, \quad (\mu_{13})_{R8,L}];$

$[(\mu_{14})_{L,R8}]$

Fig. 2.17

Partitioning of L

Step [2]: Similarly, the two MO's in L_2 are conflict free (from Fig. 2.17), hence they are both placed in I_2.

E_2 is scanned; it contains μ_3 which conflicts with μ_2 since they both use resource B. μ_3 is thus tentatively placed in the next level L_3 (in this case L_3 already contains μ_3).

Step [3]: L_3 is scanned. MO's μ_5 and μ_8 are in conflict, hence execution of one of these has to be delayed. Before this choice is made, E_3 is scanned for some MO which is conflict free with either μ_5 or μ_8. The possible candidates are μ_6 and μ_7 and since μ_7 conflicts with μ_3 (from the dependency graph) it is rejected. One the other hand μ_6 conflicts with μ_8 but is concurrently executable with both μ_3 and μ_5. Thus μ_8 is delayed. In effect, μ_6 and μ_8 are interchanged from their original partitions L_3 and L_4. The output from this step is $I_3 = \{\mu_3, \mu_5, \mu_6\}$, while μ_8 is tentatively placed in L_4.

Step [4]: Because μ_{11} is data dependent on μ_8, μ_{11} is transferred down one level to L_5. L_4 now contains $\mu_7, \mu_8, \mu_9, \mu_{10}$. L_4 is examined and it is seen that μ_7 and μ_{10} are in conflict, hence one of these must be delayed. It turns out that the choice can be either. Supposing μ_{10} to be delayed, the output produced by this step is $I_4 = \{\mu_7, \mu_8, \mu_9\}$, while L_5 contains $\mu_{10}, \mu_{11}, \mu_{12}, \mu_{13}$.

Step [5]: The dependency graph indicates μ_{10} and μ_{11} must precede μ_{12} and μ_{13} respectively. Thus an additional level, $L_5 = \{\mu_{10}, \mu_{11}\}$ is created; L_5 now contains

μ_{12}, μ_{13}. L_5 must be further partitioned into $\{\mu_{10}\}$ and $\{\mu_{11}\}$ since these MO's are in conflict. Thus microinstructions $I_5 = \{\mu_{10}\}$, $I_6 = \{\mu_{11}\}$ are obtained.

<u>Step [6]</u>: The remaining partitions L_5 and L_6 are examined. Since there are no further conflicts, the remaining microinstructions are obtained in a straightforward manner. The final form of the output is shown in Fig. 2.18

The TG algorithm is a heuristic algorithm. In particular once L has been partitioned according to resource usage, resource conflicts are resolved on the basis of heuristic rules. Unfortunately the heuristics used are not so clearly stated making an analysis of the algorithm difficult. For instance, to select MO's that have to be delayed due to resource conflicts, a rule is used that the first MO's to be delayed are those with only one successor. If additional micro-operations need to be delayed, then the delay is determined as a "function of common successors". Just what exactly this "function" is, is left unspecified.

Since resource conflict resolution does not appear to take timing into consideration, it is likely that the algorithm is applicable only to monophase systems.

Because the optimizing strategy is localized - MO's are moved from one time frame to an adjacent time frame but no further - the optimizing ability of the TG

algorithm is also limited. However, it is instructive to compare the performances of the RT, TG and JD algorithms on the same input (Fig. 2.15). The output produced by applying the RT algorithm is given by Fig. 2.19; an additional microinstruction is required. Applying the JD algorithm on the other hand, produces the same output as is produced by the TG algorithm (Fig. 18). This can be verified by examining the conflict graph that would be constructed by the JD algorithm using the information provided in Figs. 2.15 and 2.17. For the sake of simplicity (since there are no labelled edges for this particular example), the conflict graph is represented by a binary (adjacency) matrix A (Fig. 2.20):

$$A_{ij} = 1 \quad \text{if } i < j \;\&\; \sim (\mu_i || \mu_j)$$
$$= \text{otherwise.}$$

$I_1 = \{\mu_1\}$
$I_2 = \{\mu_2, \mu_4\}$
$I_3 = \{\mu_3, \mu_5, \mu_6\}$
$I_4 = \{\mu_7, \mu_8, \mu_9\}$
$I_5 = \{\mu_{10}\}$
$I_6 = \{\mu_{11}\}$
$I_7 = \{\mu_{12}, \mu_{13}\}$
$I_8 = \{\mu_{14}\}$

Fig. 2.18

Output produced by the TG algorithm

$I_1 = \{\mu_1\}$
$I_2 = \{\mu_2, \mu_4\}$
$I_3 = \{\mu_3, \mu_6\}$
$I_4 = \{\mu_5, \mu_7\}$
$I_5 = \{\mu_8\}$
$I_6 = \{\mu_9, \mu_{10}\}$
$I_7 = \{\mu_{11}\}$
$I_8 = \{\mu_{12}, \mu_{13}\}$
$I_9 = \{\mu_{14}\}$

Fig. 2.19

Output produced by the RT algorithm

	μ_1	μ_2	μ_3	μ_4	μ_5	μ_6	μ_7	μ_8	μ_9	μ_{10}	μ_{11}	μ_{12}	μ_{13}	μ_{14}
μ_1	0	1	1	1	0	0	0	1	0	0	0	0	0	0
μ_2	0	0	1	0	0	0	1	0	0	0	0	0	0	0
μ_3	0	0	0	0	0	0	1	0	0	0	0	0	0	0
μ_4	0	0	0	0	1	0	0	1	1	1	0	0	1	1
μ_5	0	0	0	0	0	1	0	1	1	1	0	0	1	1
μ_6	0	0	0	0	0	0	0	1	1	1	1	0	0	1
μ_7	0	0	0	0	0	0	0	0	0	1	1	0	1	0
μ_8	0	0	0	0	0	0	0	0	0	1	1	1	1	1
μ_9	0	0	0	0	0	0	0	0	0	1	0	1	0	0
μ_{10}	0	0	0	0	0	0	0	0	0	0	1	1	0	1
μ_{11}	0	0	0	0	0	0	0	0	0	0	0	1	1	1
μ_{12}	0	0	0	0	0	0	0	0	0	0	0	0	0	1
μ_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0	1
μ_{14}	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 2.20

Matrix Form of the Conflict Graph

2.3.4 The Yau-Schowe-Tsuchiya (YST) Algorithm [77]

Prior to describing this algorithm, a few definitions are necessary.

A *data available* MO is an MO for which all MO's on

which it is directly data dependent have been assigned to microinstructions. A set of such data available MO's is a data-available set. A complete microinstruction is a microinstruction to which no additional MO (from a data available set) can be added without causing resource conflicts.

As in the previous section, I shall describe the YST algorithm through an example. Consider the simple dependency graph of Fig. 2.21, in which the ENTRY and EXIT nodes are assumed to be such that all MO's following ENTRY are data dependent on it, and EXIT is data available only when all its preceding MO's have been executed.

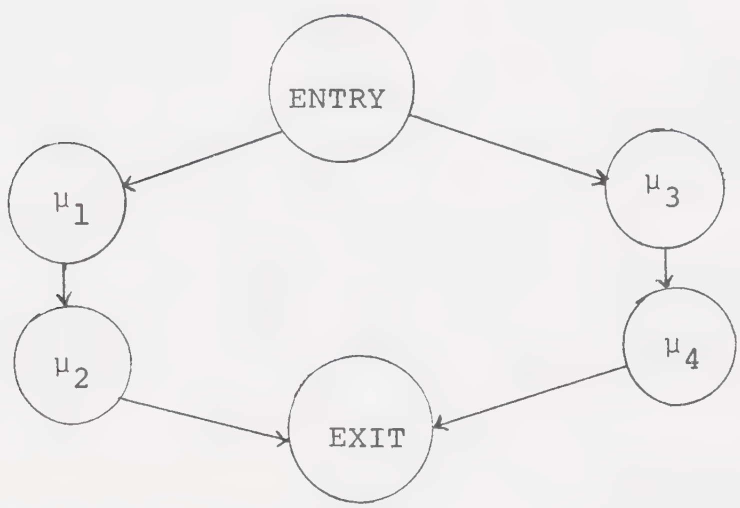

Fig. 2.21

Dependency Graph for the YST Algorithm: An Example

Detection of parallelism will then be confined to the set $M^* = \{\mu_1, \mu_2, \mu_3, \mu_4\}$. The resource (unit) conflicts between the MO's are described by a set of conflict

statements, which in terms of the notation of (2.4) are for this example:

$$U_1 \cap U_3 \neq \phi$$

$$U_2 \cap U_3 \neq \phi \qquad (2.9)$$

$$U_2 \cap U_4 \neq \phi$$

From Fig. 2.21 we observe that the lower bound on the number of microinstructions is 2. The aim of the procedure is to derive a set of alternate microinstruction sequences and select the sequence closest to this "computed" lower bound. To reduce search time, the algorithm uses several items of information to decide whether to terminate searching for a particular sequence or not. In the following description, these termination criteria are indicated informally. For further details the reader is referred to [77].

Given the dependency graph (Fig. 2.21) and the conflict statements (2.9), the YST algorithm proceeds as follows:

Step [1]: Each $\mu_i \in M^*$ is placed in a separate (temporary) partition: $I_1 = \{\mu_1\}$, $I_2 = \{\mu_2\}$, $I_3 = \{\mu_3\}$, $I_4 = \{\mu_4\}$, $P = \{I_1, I_2, I_3, I_4\}$. $|P|$ denotes the "current" upper bound on the number of microinstructions.

Step [2]: Generate the data available set D and the data

non-available set $D' = M^* - D$. Here $D = \{\mu_1, \mu_3\}$, $D' = \{\mu_2, \mu_4\}$.

Step [3]: Select from D, a complete microinstruction. Here $\{\mu_1\}$ and $\{\mu_3\}$ are possible candidates. Select any one of these arbitrarily, say $\{\mu_1\} = I_5$, and set $I \leftarrow I \cup I_5$, where I (initially empty) is the set of complete microinstructions already generated. On completing this step, $I = \{I_5\}$. The remaining elements in D are saved in a separate partition I_6.

Step [4]: D is enlarged with those MO's in D' made data available as a result of the selection in Step [3]; the same MO's are also deleted from D'. Thus $D = \{\mu_2, \mu_3\}$, $D' = \{\mu_4\}$.

Step [5]: Repeat Step [3]. Two trivial complete microinstructions $\{\mu_2\}$, $\{\mu_3\}$ are possible. $I_7 = \{\mu_2\}$ is arbitrarily selected and $I \leftarrow I \cup I_7$. On completing this step, $I = \{I_5, I_7\}$, $D = \{\mu_3\}$ and $I_8 = \{\mu_3\}$ is saved.

Step [6]: Repeat Step [4]. However since μ_4 is data dependent on μ_3 and μ_3 is still D, D and D' remain unchanged.

Step [7]: Repeat Step [3] for $D = \{\mu_3\}$. This results in $I = \{I_5, I_7, I_9\}$, where $I_9 = \{\mu_3\}$. On repeating Step [4], $D = \{\mu_4\}$, $D' = \phi$.

At this stage, since $D = \{\mu_4\}$, $D' = \phi$, $|I| = |P| - 1$, repeating Steps [3] and [4] would result in $|I| = |P|$. The algorithm stops pursuing this particular sequence since it anticipates that the number of resulting microinstructions would be $|P| = 4$. Instead:

Step [8]: It backtracks and selects $I_6 = \{\mu_3\}$ saved in the first iteration of Step [3] as an initial complete microinstruction. Note that this selection re-initialized D and D' to $D = \{\mu_1, \mu_3\}$, $D' = \{\mu_2, \mu_4\}$. On iterating Steps [3] and [4], the algorithm obtains $I = \{I_6, I_{10}, I_{11}\}$, $I_6 = \{\mu_3\}$, $I_{10} = \{\mu_1, \mu_4\}$, $I_{11} = \{\mu_2\}$. Thus $|I| < |P|$, and the output is nearer to the computed lower bound. P is set to I.

Step [9]: Since $|P| > 2$ still, and there remains a microinstruction choice saved at an earlier stage (viz., $I_8 = \{\mu_3\}$), the algorithm backtracks and selects $I_8 = \{\mu_3\}$ as a possible choice instead of $I_7 = \{\mu_2\}$. For this backtrack to be effective D and D' are re-initialized to $D = \{\mu_2, \mu_3\}$, $D' = \{\mu_4\}$. On iterating Steps [3] and [4], the output produced is $I = \{I_5, I_8, I_{12}\}$, $I_5 = \{\mu_1\}$, $I_8 = \{\mu_3\}$, $I_{12} = \{\mu_2\}$, and $D = \{\mu_4\}$. Since $|I| = |P|$ and $D = \phi$, pursuance of this sequence is stopped. Finally, since no other microinstruction choices remain, the algorithm terminates producing as its result, the output from Step [8].

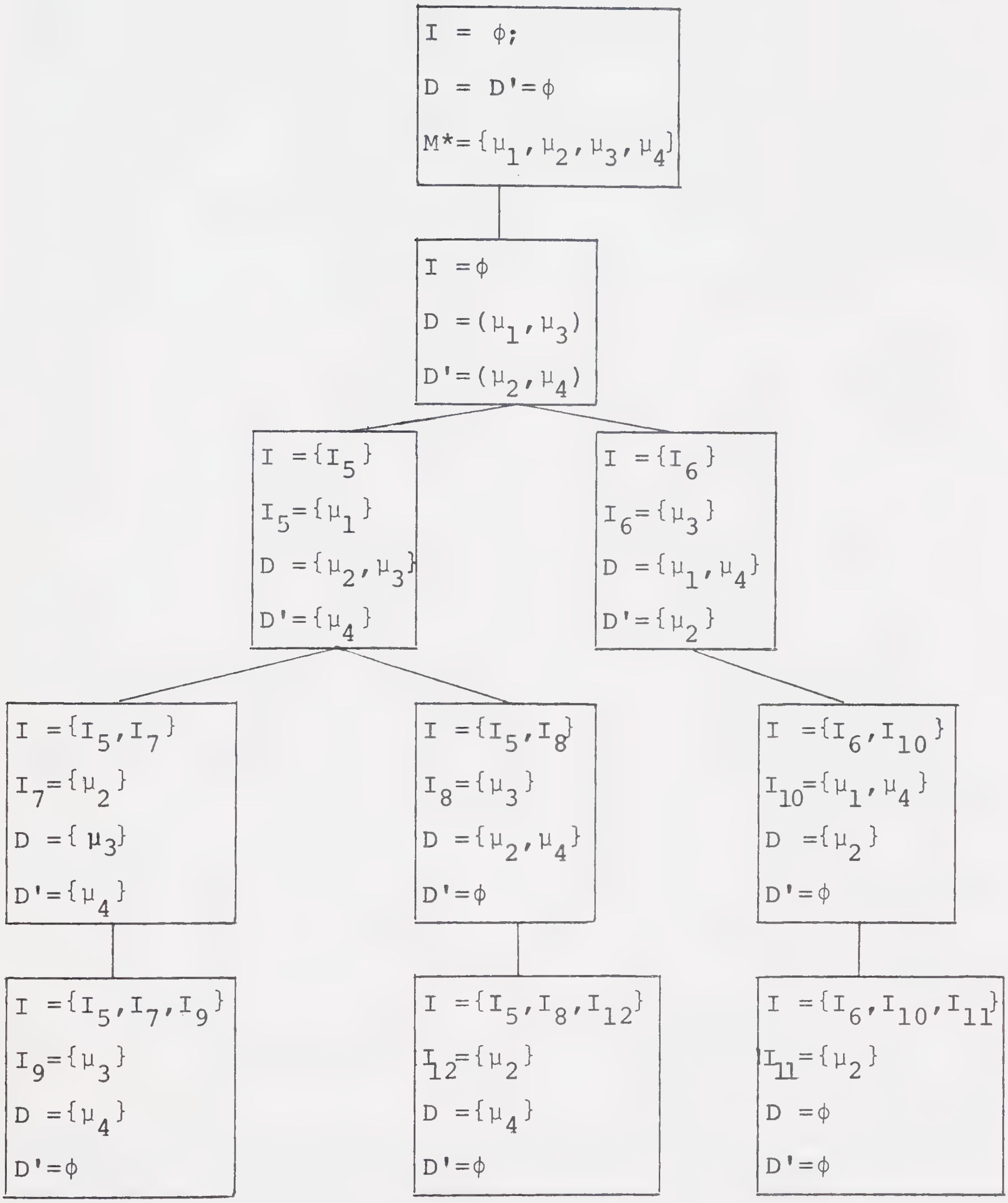

Fig. 2.22

Search Tree Generated by the YST Algorithm

A schematic view of the search tree generated by the YST algorithm is shown by Fig. 2.22. In practical situations the size of the solution space may be reduced considerably, by the data dependencies and the potential parallelism between MO's. In the given example for instance, possible sequences in the solution are $<\mu_1\mu_2\mu_3\mu_4>$, $<\mu_1\mu_3\mu_2\mu_4>$, $<\mu_3(\mu_1,\mu_4)\mu_2>$, $<\mu_3\mu_1\mu_2\mu_4>$ and $<\mu_3\mu_4\mu_1\mu_2>$. Of these, the last two are never generated by the search process because once μ_3 is selected as the first complete microinstruction, μ_1 and μ_4 will always be placed together as parallel MO's.

In the worst case however, the number of nodes generated in the search tree will be exponential in n (the length of the SLM). Since each node will necessitate at least one pairwise comparison between MO's, the worst case complexity of the algorithm will be $O(K^n)$ for some K.

Such a situation arises for example with the following sequence of MO's:

$$\begin{array}{llll} \mu_1 : & A \leftarrow B + C & U_1 \cap U_2 \neq \phi & \\ \mu_2 : & D \leftarrow E \wedge F & U_1 \cap U_3 \neq \phi & (2.10) \\ \mu_3 : & G \leftarrow B - E & U_2 \cap U_3 \neq \phi \,. & \end{array}$$

Here μ_1,μ_2,μ_3 are mutually data independent but because of the unit conflicts, the YST algorithm will generate all 3! sequences of MO's.

A heuristic modification proposed by the authors to reduce search time is to attach a weight $w(\mu_i)$ to each

vertex μ_i in the dependency graph; $w(\mu_i)$ equals the number of MO's that are data-dependent on μ_i. Furthermore, the MO's in D are ordered according to the input sequence. Complete microinstructions are then generated starting with the first MO in D, the second MO, etc., and a <u>weight</u> $w(I_j)$ is assigned to each such microinstruction generated, this being defined by

$$w(I_j) = \sum_{\mu_i \varepsilon I_j} w(\mu_i) \ . \qquad (2.11)$$

The selection criterion in Step [3] becomes (instead of an arbitrary selection) that microinstruction with the largest weight, the rationale being that the selection will probably free the largest number of data dependent MO's for subsequent selection.

The YST algorithm being an exhaustive search procedure, guarantees an optimal sequence of microinstructions. When the heuristic method is used, optimality may not result. Finally, the algorithm ignores problems of timing.

2.4 Summary

This concludes the review of algorithms for detecting parallelism in SLM's. To summarize, the JD algorithm appears to be the most useful from the viewpoint of generality and algorithmic complexity. The YST algorithm - within the context of monophase systems - guarantees a minimal output (note that the JD method if applied to the

example of Fig. 2.21 and (2.9) will not produce an optimal output). It is however potentially inefficient.

The TG algorithm is also applicable only to mono-phase systems, uses heuristics and attempts but does not guarantee, optimal output. The RT algorithm is less general than the JD method, and may produce a lengthier output than the latter.

CHAPTER III

POTENTIAL PARALLELISM IN MICROPROGRAMMABLE PROCESSORS

3.1 Introduction

The algorithms described in Chapter II serve to determine or "expose" the parallelism within SLM's. This parallelism originates fundamentally, in the fact that within a machine's data flow, several operational units and data paths may be concurrently active without any mutual resource conflicts. The primary objective in utilizing a horizontal microword organization is to take explicit advantage of this data flow characteristic [58,71].

I shall use the term <u>potential parallelism</u> to denote this general characteristic of parallelism as embodied in a horizontal microword organization; the <u>degree of potential parallelism</u> D_p is defined as the maximum number of MO's that can be executed from a single microword.

In the present analysis, I shall assume that the control memory (CM) is a regular array in the sense that all its words are identically organized (Fig. 3.1). Thus given a CM with y microwords $W_1, W_2, \ldots, W_y$, D_p is the same for all W_i $(i = 1, \ldots, y)$.

A microinstruction I_j, is then nothing but a particular <u>state</u> of a microword say W_k, in the sense that a

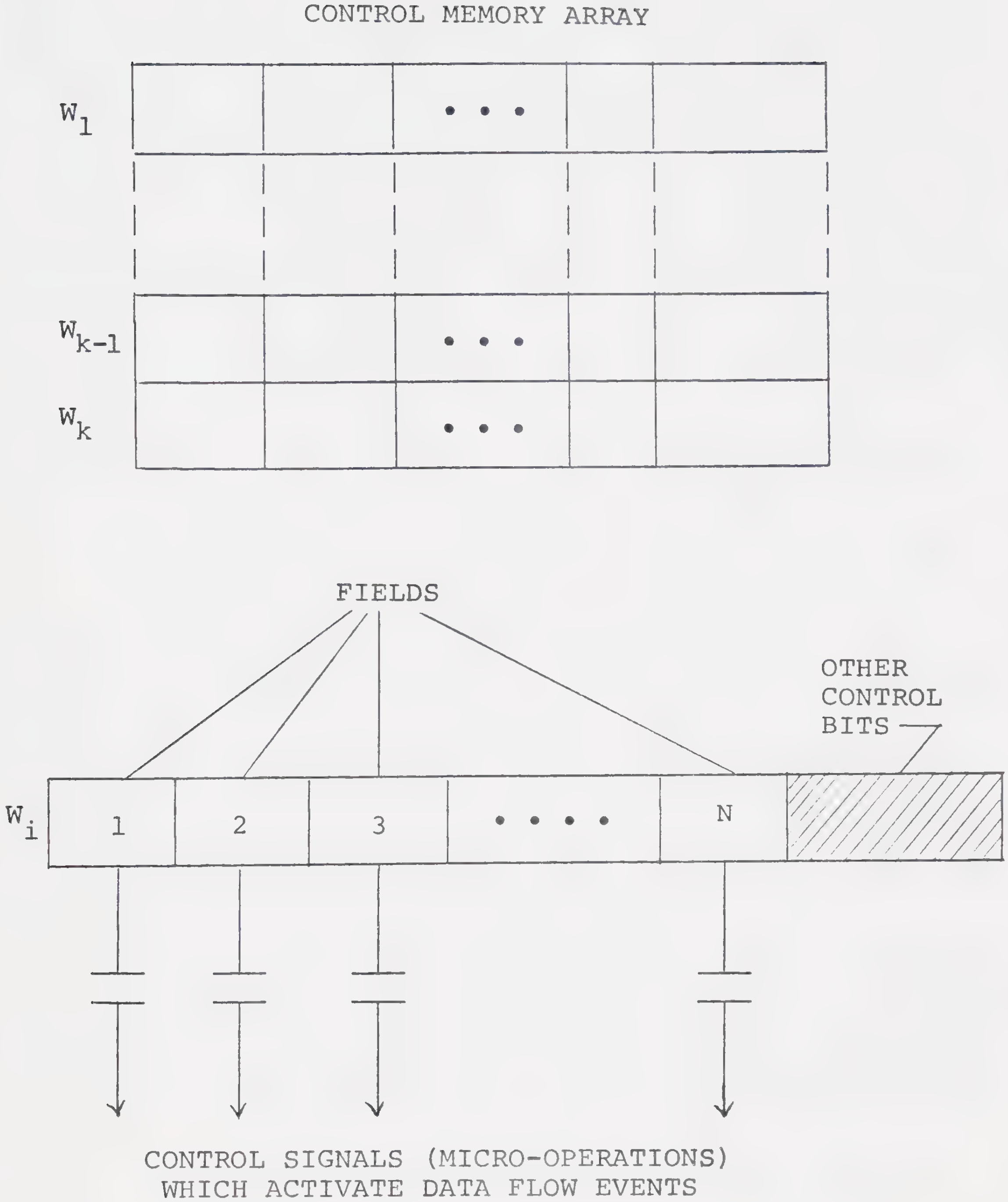

Fig. 3.1

Control Memory and Typical Control Memory Word Organization

specific subset of MO's have been specified (by the microprogrammer) to be executed from W_k. W_k can thus also be viewed as a state variable whose individual values, the states, denote possible microinstructions that can be stored in W_k. Given a microinstruction I_j, the <u>degree of actual parallelism</u> of I_j, D_a (I_j) is simply the cardinality of the MO set comprising I_j. Thus while D_p is invariant for some given microword organization, D_a may (and in general, will) vary from one microinstruction to another (Fig. 3.2). However, D_p denotes an upper bound on D_a.

As I pointed out in Chapter I, potential parallelism becomes significant as a concept in the context of writable control memories (WCM's). In designing a microword organization for a WCM, the nature of the user microprograms will not be known to the designer. Thus, enhancing the microword potential parallelism is clearly one of the most important feasible design objectives.

The purpose of this chapter is to examine the formal nature of potential parallelism and a means of maximizing it; and to analyse its effect on the control memory word size.

3.2 Analysis of Potential Parallelism

Let $\mu^* = \{\mu_1, \mu_2, \ldots, \mu_q\}$ denote the set of all MO's in a microprogrammable processor. Then informally, $\mu_i, \mu_j \in \mu^*$ are defined to be potentially parallel, denoted

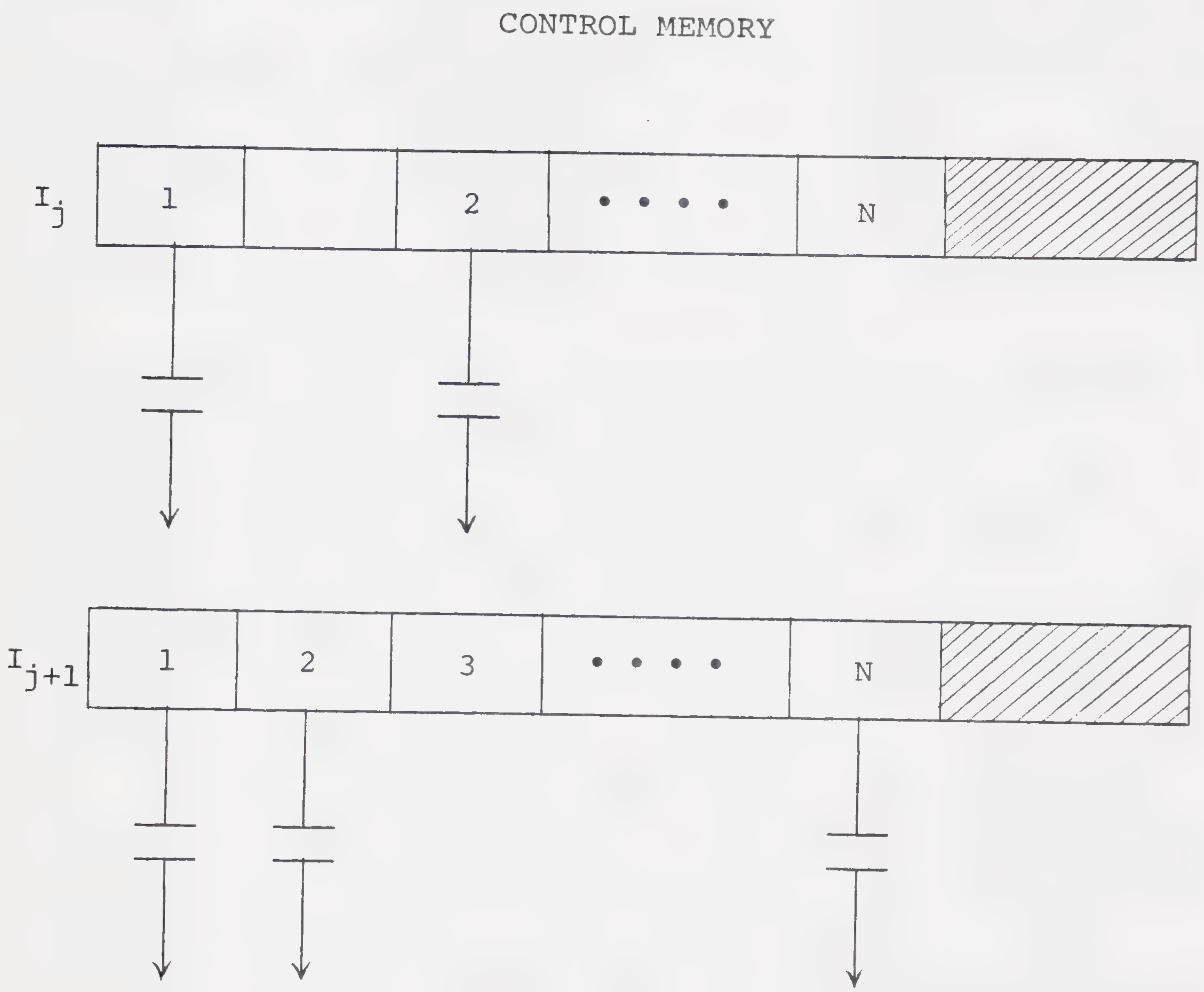

Fig. 3.2

Two Possible Microinstruction Configurations

$\mu_i ||_p \mu_j$ if their executions involve no conflicts between their resources. More formally, defining the resource independent relation σ as

$\mu_i \sigma \mu_j$ if $(\mu_i \beta \mu_j) \wedge (U_i \cap U_j = \phi)$ for $\mu_i, \mu_j \varepsilon \mu^*$, then $\mu_i ||_p \mu_j$ if

$$[V_i \cap V_j = \phi] \vee [(V_i \cap V_j \neq \phi) \wedge (\mu_i \sigma \mu_j)] \qquad (3.1).$$

Notice that if $V_i \cap V_j = \phi$, μ_i, μ_j will always be executed in separate clock cycle phases, hence there will be no resource conflicts. On the other hand, the condition $(V_i \cap V_j \neq \phi) \wedge (\mu_i \sigma \mu_j)$ means that though μ_i, μ_j are executed in the same phase, they are conflict free since they use disjoint resource sets.

Recall from (3.1), the quantity D_p. The problem of interest here is to maximize D_p. From (3.1), note that the $||_p$ relation between μ_i, $\mu_j \varepsilon \mu^*$ is determined by their respective resource sets R_i, R_j and time validities V_i, V_j. Suppose for some pair μ_i, μ_j, $\sim(\mu_i ||_p \mu_j)$. Then evidently $(V_i \cap V_j \neq \phi) \wedge \sim(\mu_i \sigma \mu_j)$. More particularly, assuming that time validities have <u>not</u> been (yet) assigned to MO's, then for a pair of MO's μ_i, μ_j such that $\sim(\mu_i \sigma \mu_j)$, if we could assign the time validities V_i, V_j such that $V_i \cap V_j = \phi$, we would force μ_i, μ_j to become potentially parallel.

I shall call this, the <u>phase</u> <u>allocation</u> problem. Note that it implies a basic assumption: that the machine

cycle follows a polyphase timing scheme. However, the approach developed here can also be used to assess the feasibility of polyphase schemes - an aspect which I shall further discuss later. The phase allocation problem is stated more precisely as follows:

Let μ^* be a set of q MO's with equal execution times t_m. Determine and allocate a minimal k-phase clock cycle

$$C = \langle \Pi_1, \Pi_2, \ldots, \Pi_k \rangle \tag{3.2}$$

where $\Pi_i \cap \Pi_j = \phi$ for $i \neq j$, $t(\Pi_i) = t$ for all i, $t(\Pi_i)$ denoting the duration of phase Π_i, and $t > t_m$, such that the degree of potential parallelism $D_p = q$.

The objective then, is to make all q MO's in μ^* pairwise potentially parallel. Observe that $D_p = q$ is obtained trivially by allocating each $\mu_i \in \mu^*$ to a separate phase Π_i. In that case $k = q$ and we obtain a q-phase cycle. This solution is neither minimal nor practical.

The procedure developed below uses the following heuristics:

(a) All MO's utilizing the same operational unit are to be allocated to the same phase.

(b) All MO's which can be allocated to the same phase without violating the $||_p$ relation will be so allocated.

(c) Pairs of MO's not satisfying (a) or (b) will be allocated to disjoint phases.

Rule (a) is essentially a "realistic" hardware constraint. For, let

$$\begin{aligned} \mu_i &= \langle OP_i, SC_i, SK_i, U_i, ? \rangle \\ \mu_j &= \langle OP_j, SC_j, SK_j, U_j, ? \rangle \end{aligned} \qquad (3.3).$$

be a pair of MO's with unspecified time validities (denoted by '?') using the same unit U; clearly if they were to be assigned the same time validities then $\sim(\mu_i \,||_p\, \mu_j)$. On the other hand if they were assigned disjoint time validities say $V_i = \Pi_1$, $V_j = \Pi_2$, then when μ_i is executed, U will be activated in phase Π_1, and when μ_j is executed, U would be activated in Π_2. In theory there is no restriction on such a timing mechanism. In practice the complexity of the resulting circuitry would be prohibitive, hence the imposition of rule (a).

In order to apply this rule, consider the set of MO's μ^*. Partition μ^* into a disjoint subsets $\bar{\mu}_1, \bar{\mu}_2, \ldots, \bar{\mu}_n$ such that for any $\bar{\mu}_i$, $\mu_j, \mu_k \varepsilon \bar{\mu}_i$ utilize the same operational unit. Call such a set, a <u>unit equivalent set</u>. An example of such a set is:

$$\begin{aligned} \mu_1 &= \langle \text{ADD}, \{\text{M1}, \text{M2}\}, \{\text{M3}\}, \{\text{ADDER}\}, ? \rangle \\ \mu_2 &= \langle \text{SUB}, \{\text{M1}, \text{M2}\}, \{\text{M4}\}, \{\text{ADDER}\}, ? \rangle \\ \mu_3 &= \langle \text{ADD}, \{\text{M3}, \text{M4}\}, \{\text{M4}\}, \{\text{ADDER}\}, ? \rangle \end{aligned} \qquad (3.4).$$

Given a unit equivalent set $\bar{\mu}_i = \{\mu_{i1}, \mu_{i2}, \ldots, \mu_{i,k_i}\}$, a

unit equivalent MO is defined by the 5-tuple

$$< OP_i, SC_i, SK_i, U_i, ? > \qquad (3.5)$$

where $OP_i = \bigcup_{j=1}^{k_i} OP_{ij}$, $SC_i = \bigcup_{j=1}^{k_i} SC_{ij}$, $SK_i = \bigcup_{j=1}^{k_i} SK_{ij}$,

$$U_i = U_{ij} \; (j = 1,\ldots,k_i)$$

For example, given the set (3.4), the corresponding unit equivalent MO is

$$< \{ADD, SUB\}, \{M1, M2, M3, M4\}, \{M3, M4\}, \{ADDER\}, ? > \qquad (3.6).$$

Thus, from the original set μ^*, we can obtain a set of q^* unit equivalent MO's, μ_E^*:

$$\mu_E^* = \{\mu_1, \mu_2, \ldots, \mu_{q^*}\} \qquad (3.7).$$

Henceforth I shall simply refer to members of μ_E^* as "MO's", the prefix "unit equivalent" being implicitly understood. Furthermore, whatever time validity is assigned to some $\mu_i \in \mu_E^*$, will imply the assignment of the same time validity to all members of the corresponding unit equivalent set represented by μ_i. This procedure then, completes the implementation of rule (a).

In the rest of this section, I shall continue to represent an MO by (2.4) except that the V field is left undefined. The problem of course, is to determine these V fields for each MO in μ_E^*.

A subset $\mu' \subseteq \mu_E^*$ is termed a resource independent class (RC) if for all pairs $\mu_i, \mu_j \in \mu'$, $\mu_i \sigma \mu_j$. A maximal RC (MRC) is an RC to which no other MO can be added without violating the σ relation. Given μ_E^*, a set of MRC's can then be constructed. Denote this set by

$$\rho = \{\rho_1, \rho_2, \ldots, \rho_n\} \tag{3.8}$$

Thus each ρ_i is a set of MO's that are pairwise resource independent. By (3.1) they are therefore pairwise potentially parallel even if assigned to the same phase. Following rule (b) then, an MRC can be allocated the same phase.

Example 3.1

Suppose μ_E^* contains 8 MO's, denoted $\mu_1, \ldots, \mu_8$. Let the MRC's as determined by applying the above definitions be:

$$\rho' = \{\rho_1', \rho_2', \rho_3', \rho_4', \rho_5', \rho_6'\} \tag{3.9}$$

where

$$\begin{aligned} \rho_1' &= \{\mu_1, \mu_2, \mu_3\} & \rho_4' &= \{\mu_5, \mu_7, \mu_8\} \\ \rho_2' &= \{\mu_2, \mu_3, \mu_5\} & \rho_5' &= \{\mu_4, \mu_8\} \\ \rho_3' &= \{\mu_4, \mu_6\} & \rho_6' &= \{\mu_6, \mu_7\} \end{aligned} \tag{3.10}$$

Note that the MRC's are not necessarily disjoint. For instance μ_3 belongs to both ρ_1' and ρ_2'.

A <u>cover</u> (or <u>covering set</u>) θ is a set of MRC's such that (i) all the MO's in μ_E^* are included in θ; (ii) no MO appears in more than one MRC; and (iii) if any of the MRC's (or their subclasses) are deleted from θ, one or more MO's will be excluded. A <u>minimum cover</u> θ_{min} is a cover containing the smallest number of MRC's (or their subclasses).

Given a set ρ of MRC's, covers can be systematically discovered by applying one of several well known methods used for the simplification of switching functions [19,41], minimizing incompletely specified sequential machines [41], or minimizing control memory word dimensions [20] (See for example, the next section). Thus, for Example 3.1 the following covers are obtained:

$$\theta_1 = (\rho_1', \rho_3', \rho_4') = (\overline{\mu_1, \mu_2, \mu_3}; \overline{\mu_4, \mu_6}; \overline{\mu_5, \mu_7, \mu_8})$$

$$\theta_2 = (\rho_1', \rho_2', \rho_3', \rho_5', \rho_6') = (\overline{\mu_1, \mu_2, \mu_3}; \overline{\mu_5}; \overline{\mu_4, \mu_6}; \overline{\mu_8}; \overline{\mu_7}) \quad (3.11).$$

$$\theta_3 = (\rho_1', \rho_4', \rho_5', \rho_6') = (\overline{\mu_1, \mu_2, \mu_3}; \overline{\mu_5}, \overline{\mu_7, \mu_8}; \overline{\mu_4, \mu_6})$$

The minimum cover θ_{min} for this example, is of course θ_1.

It was stated earlier that members of an MRC are pairwise resource independent and therefore potentially parallel. If a pair of MO's μ_k, μ_l being to <u>disjoint</u> MRC's say ρ_i, ρ_j (i.e., $\rho_i \cap \rho_j = \phi$) then μ_k, μ_l are <u>not</u> resource independent; they can only be potentially parallel if assigned to disjoint phases. A cover θ signifies that

if each MRC (or its subclass) in θ is assigned to a distinct phase then the MO's in θ will <u>all</u> be pairwise potentially parallel (this proposition is proved below). The minimum cover θ_{min} will then determine the smallest number of machine cycle phases that preserves this parallelism.

<u>Theorem 3.1</u>

If q^* is the number of MO's in μ_E^*, then any cover θ gives a value of $D_p = q^*$ if the MRC's (or their subclasses) in θ are assigned to distinct machine cycle phases.

<u>Proof</u>

Let a cover $\theta = (\rho_1, \rho_2, \ldots, \rho_t)$. By definition $\mu_{i1}, \mu_{i2}, \varepsilon \rho_i$ satisfy $\mu_{i1} \sigma \mu_{i2}$. Thus if the MO's in ρ_i are assigned to the same phase say Π_i, then they are pairwise potentially parallel (by 3.1); i.e. $\mu_{i1} \,||_p\, \mu_{i2}$ for all $\mu_{i1}, \mu_{i2} \,\varepsilon\, \rho_i$.

If $|\rho_i|$ denotes the cardinality of ρ_i, the value of D_p for ρ_i is

$$D_p^i = |\rho_i|, \quad i = 1, \ldots, t \,. \tag{3.12}$$

If each ρ_i is assigned to a distinct phase Π_i of some clock cycle C, then $V_{i1} \cap V_{j1} = \phi$ for $\mu_{i1} \,\varepsilon\, \rho_i$, $\mu_{j1} \,\varepsilon\, \rho_j$, implying that $\mu_{i1} \,||_p\, \mu_{j1}$ for all $\mu_{i1} \,\varepsilon\, \rho_i$, $\mu_{j1} \,\varepsilon\, \rho_j$.

Finally, let each MO be assigned to a distinct field in the microword. Then all q^* MO's may be executed from a

single microword without resource conflicts. Thus $D_p = q^*$. □

In example (3.1), the minimum cover $\theta_{min} = (\rho_1', \rho_3', \rho_4')$ requires a 3-phase cycle (i.e., k = 3):

$$C = (\Pi_1, \Pi_2, \Pi_3) .,$$

Thus if the MO's in ρ_1' are assigned to Π_1, those in ρ_3' to Π_2, and those in ρ_4' to Π_3, $D_p = q^* = 8$, provided the MO's are assigned to distinct fields in the WCM word.

The solution to the phase allocation problem results in a microword that exhibits the maximum possible value of D_p - subject of course to the fact that the MO's are unit equivalent MO's. Whether this solution is practical will depend among other factors, on the value of k (the number of phases obtained) and on the significance (or importance) of parallelism within the overall set of design objectives.

At the design level, the problem of deciding the length (duration) of the machine cycle, and the number of its component phases is quite complex since several factors may affect the decision. A discussion of the pragmatics underlying such timing decisions is provided by Langdon [45]. Here, I shall consider only one of these aspects, viz., the relationship between the machine cycle and the cycle time of CM. As Langdon points out, the latter has a large influence on deciding the length of the machine cycle.

Let the CM cycle time be T_{cm}, and k the minimum number of phases of length t obtained as a solution to the

phase allocation problem. The machine cycle C would then be of length kt. If $kt \leq T_{cm}$, the lower bound on the machine cycle would in any case, be T_{cm}. Thus a k-phase cycle can be utilized and a maximum value $D_p = q^*$ be preserved. On the other hand, if $kt > T_{cm}$ the above allocation scheme may unduly increase the <u>effective</u> CM cycle time to kt. If the increase is too high, then of course, much of the advantage of a highly parallel microword and a fast CM would be lost.

3.3 Minimization of the Word Length of WCM's

The foregoing analysis was concerned with maximizing potential parallelism. The reader will note (from the proof of Theorem 3.1) that each (unit equivalent) MO must be assigned to a distinct field of the microword in order that $D_p = q^*$.

The resulting microword organization is one where each unit equivalent set (of MO's) is assigned (i.e., encoded by) a distinct field.

A problem that is in a sense, dual to the potential parallelism maximization problem, is that of minimizing the <u>word length</u> of control memories. This problem has been studied by several people, notably by Schwartz [62], Grasselli and Montanari [32], and Das <u>et al</u> [20]. Implicit in these investigations were the following two assumptions:

Assumption (a):

A set of control memory words W_1, W_2, ..., W_y are already available, each word containing one or more MO's (Fig. 1.2). That is, a read-only memory with a direct control word organization [38] is given. The problem is one of determining a minimally encoded organization [58] such that the microword bit dimension is minimized.

Assumption (b):

The problem solution ignores the condition where two MO's can only be activated in two different clock cycle phases and as a consequence, may not be grouped into the same field of the microword. In other words, polyphase microinstructions are not considered.

Using the conditions for parallelism in SLM's (2.7), Dasgupta and Tartar extended the method of Das *et al* to the case of ROM minimization for polyphase schemes [23]. The assumption made in [23] was that time validities were already assigned to MO's prior to the specification of read-only microprograms.

In the present section, I shall consider the problem of minimizing the word length ("bit dimension") of *writable control memories*. Recall that in designing WCM's, no knowledge is available concerning the microprograms that will be stored in the memory. Hence ROM minimization techniques cannot be directly applied here.

In the present analysis, it is assumed that MO's are completely specified. Let $\mu^* = \{\mu_1, \mu_2, \ldots, \mu_q\}$ be this set of MO's with time-validities assigned. Then a <u>potential compatibility class</u> (PCC) is a set of MO's such that for all pairs μ_i, μ_j in the PCC, $\sim(\mu_i \,||_p\, \mu_j)$. A <u>maximal potential compatibility class</u> (MPCC) is simply a PCC to which no other MO can be added without violating the "$\sim||_p$" relation.

Clearly, any pair $\mu_i, \mu_j \in \mu^*$ such that $V_i \cap V_j = \phi$, can never belong to the same MPCC. For, by definition of the $||_p$ relation (3.1), $V_i \cap V_j = \phi$ implies $\mu_i \,||_p\, \mu_j$, so they can never belong to the same PCC, hence to the same MPCC. In determining MPCC's, this fact can be used to reduce slightly the computational time.

For the set μ^*, we can obtain a set of MPCC's. Each MPCC thus identifies a set of MO's that cannot be activated together in a microinstruction because of resource conflicts. Furthermore all members of an MPCC have the same (or overlapping) time validities. If the clock cycle phases are non-overlapping, then members of an MPCC will all have the same time validity, hence they can be placed within a single microword field and be activated by the same clock signal. The remainder of this analysis thus assumes a polyphase, non-overlapping timing scheme.

The WCM minimization problem can now be stated precisely as follows: let the set of MPCC's corresponding to

μ^* be denoted by $\phi = \{\phi_1, \phi_2, \ldots, \phi_s\}$, where

$$\phi_i = \{\mu_{i1}, \mu_{i2}, \ldots, \mu_{i,k_i}\}, \quad i = 1, \ldots, s \qquad (3.13).$$

Then, the problem is to find a set $\phi^* = \{\phi_1^*, \ldots, \phi_n^*\}$ of PCC's such that (i) every MO in μ^* is in at least one PCC of ϕ^*; and (ii) the quantity

$$B = \sum_{h=1}^{n} \lceil \log_2(|\phi_h^*| + 1) \rceil \qquad (3.14)$$

is minimal, where $|\phi_h^*|$ denotes the cardinality of ϕ_h^* (1), and $\lceil I \rceil$ denotes the <u>least</u> integer $\geq$ I. The quantity B designates the <u>cost</u> of the minimal cover.

Since a PCC is a collection of MO's whose executions are mutually exclusive, these MO's cannot belong to the same microinstruction. In this sense a PCC (MPCC) is equivalent to the CC (MCC) of Das <u>et al</u> [20]. Hence the minimization technique developed in [20] can be followed for the WCM problem, once the MPCC's are obtained. For the sake of completeness, this procedure is outlined below.

Given a set of MO's μ^*, and a set of MPCC's ϕ, a table, called the <u>WCM cover table</u> is constructed, by specifying the MO's, $\mu_1, \ldots, \mu_k$ in a row, and by entering ϕ_j below μ_i if $\mu_i \in \phi_j$. Each column of the table is therefore a collection of those MPCC's that contain the specified MO. Note the analogy of the WCM cover table with

---

(1) A 1 is added to $|\phi_h^*|$ to include the "NO-OP" MO in each field.

the cover table used in simplifying switching functions [19].

Example 3.2

Suppose the set of MPCC's for some specific collection of MO's is as shown in Fig. 3.3. Then the corresponding WCM cover table is given by Fig. 3.4.

$\phi_1 = \{\mu_1, \mu_7, \mu_{11}\}$ $\qquad$ $\phi_6 = \{\mu_5, \mu_7, \mu_{10}, \mu_{11}\}$

$\phi_2 = \{\mu_2, \mu_7, \mu_{11}\}$ $\qquad$ $\phi_7 = \{\mu_5, \mu_8\}$

$\phi_3 = \{\mu_3, \mu_{10}, \mu_{11}\}$ $\qquad$ $\phi_8 = \{\mu_5, \mu_9, \mu_{11}\}$

$\phi_4 = \{\mu_4, \mu_7, \mu_{10}\}$ $\qquad$ $\phi_9 = \{\mu_6, \mu_7, \mu_{10}, \mu_{11}\}$

$\phi_5 = \{\mu_4, \mu_9\}$ $\qquad$ $\phi_{10} = \{\mu_6, \mu_9, \mu_{11}\}$

Fig. 3.3

Maximal Potential Compatible Classes

The MPCC's appearing alone in some columns of the cover table are called globally essential, and the corresponding MO's heading these columns are called globally distinguished MO's; these are identified by asterisks in the cover table (see Fig. 3.4). The corresponding columns are also called globally essential.

A solution ϕ^* of a WCM cover table is a set of

μ_{*1}	μ_{*2}	μ_{*3}	μ_4	μ_5	μ_6	μ_7	μ_{*8}	μ_9	μ_{10}	μ_{11}
ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_6	ϕ_9	ϕ_1	ϕ_7	ϕ_5	ϕ_3	ϕ_1
			ϕ_5	ϕ_7	ϕ_{10}	ϕ_2		ϕ_8	ϕ_4	ϕ_2
				ϕ_8		ϕ_4		ϕ_{10}	ϕ_6	ϕ_3
						ϕ_6			ϕ_9	ϕ_6
						ϕ_9				ϕ_8
										ϕ_9
										ϕ_{10}

Fig. 3.4

WCM Cover Table

MPCC's (or their subclasses) such that (i) ϕ^* contains all the MO's in μ^*; and (ii) if any of the MPCC's (or their subclasses) in ϕ^* is omitted, at least one MO is not included. A solution is minimal if the cost B is minimum.(2)

Intuitively, a solution ϕ^* signifies that each MPCC (or a subclass of an MPCC) in ϕ^* is representable by a single encoded field in the microword. Clearly the best (minimal) solution will be that requiring the least number of bits to encode all the fields.

(2) Note the analogy between a "solution" and a "cover" as defined in section 3.2. In fact covers can be derived in precisely the same manner as solutions are derived here.

If column i of a WCM cover table forms a proper subset of some other column j, then column i <u>dominates</u> column j [20].

Consider a WCM cover table containing a globally essential column say i, and let the corresponding globally essential MPCC be ϕ_j. Then ϕ_j must appear in a solution ϕ^* (since ϕ_j is the only MPCC containing μ_i). If column i dominates column j, the latter may be deleted from the table since μ_j is contained in the MPCC in column i. Similarly, a column being dominated by a non-essential column may also be deleted. Finally, if two or more columns are exactly identical all but one of these may be deleted.

Given a WCM cover table, if its dominated columns are deleted, and the essential columns removed, the resulting table is a <u>reduced</u> WCM cover table.

For example, in Fig. 3.4, ϕ_1, ϕ_2, ϕ_3 and ϕ_7 are globally essential; columns 1, 2, 3 and 7 are thus also globally essential. Removing (selecting) these columns and deleting all the columns dominated by these columns yields the reduced WCM cover table of Fig. 3.5.

The solutions from a WCM cover table can be systematically found using the procedure for finding the prime implicant covers of switching functions. Thus, from the reduced table of Fig. 3.5, the solutions

μ_4	μ_6	μ_9
ϕ_4	ϕ_9	ϕ_5
ϕ_5	ϕ_{10}	ϕ_8
		ϕ_{10}

Fig. 3.5

Reduced WCM Cover Table

obtained are $\{\phi_4, \phi_{10}\}$, $\{\phi_4, \phi_8, \phi_9\}$, $\{\phi_5, \phi_{10}\}$, and $\{\phi_5, \phi_9\}$. Combining the globally essential MPCC's with these, the complete solutions obtained are

$\{\phi_1, \phi_2, \phi_3, \phi_4, \phi_7, \phi_{10}\}$, $\{\phi_1, \phi_2, \phi_3, \phi_4, \phi_7, \phi_8, \phi_9\}$

$\{\phi_1, \phi_2, \phi_3, \phi_5, \phi_7, \phi_{10}\}$, and $\{\phi_1, \phi_2, \phi_3, \phi_5, \phi_7, \phi_9\}$.

A minimal solution is obtained from the set of solutions by means of the following procedure.

Given a solution say ϕ_1^*, a cover table (called the solution WCM cover table) is constructed with only those MPCC's in ϕ_1^*. In this table, in addition to the globally essential MPCC's, some locally essential MPCC's may also be present. These are identified by asterisks above the corresponding (locally) distinguished MO's.

Referring to the solution WCM cover table for the solution $\phi_1^* = \{\phi_1, \phi_2, \phi_3, \phi_4, \phi_7, \phi_{10}\}$ (Fig. 3.6), it can

be seen that μ_7, μ_{10}, μ_{11} can be

μ_1^*	μ_2^*	μ_3^*	μ_4^*	μ_5^*	μ_6^*	μ_7	μ_8^*	μ_9^*	μ_{10}	μ_{11}
ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_7	ϕ_{10}	ϕ_1	ϕ_7	ϕ_{10}	ϕ_3	ϕ_1
						ϕ_2			ϕ_4	ϕ_2
						ϕ_4				ϕ_3
										ϕ_{10}

Fig. 3.6

Solution WCM Cover Table

covered by more than one MPCC. To find all possible ways of covering these MO's, a reduced solution WCM cover table containing columns 7, 10, 11 is constructed (Fig. 3.7), and all the solutions from this are obtained as: $\{\phi_1, \phi_3\}$, $\{\phi_2, \phi_3\}$, $\{\phi_2, \phi_4\}$, $\{\phi_3, \phi_4\}$, $\{\phi_4, \phi_{10}\}$, $\{\phi_1, \phi_4\}$.

μ_7	μ_{10}	μ_{11}
ϕ_1	ϕ_3	ϕ_1
ϕ_2	ϕ_4	ϕ_2
ϕ_4		ϕ_3
		ϕ_{10}

Fig. 3.7

Reduced Solution WCM Cover Table

If say $\{\phi_4, \phi_{10}\}$ is used to cover $\mu_7, \mu_{10}, \mu_{11}$, clearly the appearance of these MO's can be deleted from all other MPCC's. This results in the solution

$$\{\mu_1\}, \{\mu_2\}, \{\mu_3\}, \{\mu_5, \mu_8\}, \{\mu_4, \mu_7, \mu_{10}\}, \{\mu_6, \mu_9, \mu_{11}\}$$

whose cost, computed according to (3.14), is 9. The procedure is repeated for the other solutions obtained from the reduced solution cover table corresponding to ϕ_1^*. Similarly, starting with $\phi_2^*, \phi_3^*, \ldots, \phi_n^*$, solutions can be obtained; the one giving the smallest value of B is the minimal solution.

3.4 Conclusions

The purpose of this chapter was to examine the nature of parallelism between MO's and its relationship to two basic design problems; the constructions of polyphase timing schemes, and minimally encoded microword organizations. These problems are pertinent in both microprogrammed (with ROM's) and microprogrammable (with WCM's) systems. I have considered here the latter problem, hence the stress on "potential" rather than "actual" parallelism in this chapter.

Suppose a design process begins with maximizing potential parallelism using the procedure of section 3.2, and the number of phases obtained is k. If k is "acceptable" from the economic viewpoint, then the maximum

potential parallelism (say q*) is preserved by encoding each set of unit equivalent MO's by a single field; in effect q* fields are obtained. Clearly, subsequent application of the word minimization procedure of section 3.3 will be unnecessary since it will not reduce the microword length any further. On the other hand, if phase allocation is such that less than the maximum potential parallelism is obtained (this will happen if $k' < k$ phases are used) then microword minimization procedure may be effective. There is therefore, in this sense, a tradeoff between potential parallelism and microword length.

CHAPTER IV

PARALLELISM IN STRAIGHT LINE MICROPROGRAMS

4.1 Introduction

In Chapter II, I have reviewed several algorithms that detect parallel micro-operations in SLM's. The principle conclusions were that the JD algorithm is distinguishable as being the most general in its applicability to different host machine structures; it is also quite efficient. The YST algorithm on the other hand, produces an optimal (i.e. minimal) output for monophase microprograms, ignores timing considerations, and is asymptotically inefficient. The other two algorithms are inferior to these either in terms of generality or optimality.

In the present chapter, the problem of optimizing parallelism in SLM's is considered at a greater level of generality than has been done hithertofore. In particular, the idea of permuting the input sequence (SLM), a technique used by both Tsuchiya and Gonzalez [72] and Yau *et al* [77] though in a limited way, is explored and analysed more systematically within a polyphase framework.

The concrete result of this analysis is a new, efficient, optimizing algorithm which is applicable to both monophase and polyphase systems. The algorithm

produces an output which, though n ot optimal, is the 'smallest' in a more restricted sense.

4.2 Basis for the Optimizing Algorithm

A useful concept that needs be introduced at this point, is that of a (microprogrammable) machine state. This is defined as the outcome of an assignment of values to each distinct memory resource in the machine. Each memory resource can itself be regarded as a state variable taking values from a well-defined range. Thus, it also makes sense to talk of the state of a subset of memory elements. The overall machine state is then given by the ordered set of values assumed by the memory resources at that time.

As a trivial example, if $\{M_1,M_2,M_3,M_4\}$ is the set of memory elements in a machine, then the set of values $(M_1 = 3, M_2 = 6, M_3 = 1, M_4 = 0)$ defines a state that is distinct from the state defined by the values $(M_1 = 6, M_2 = 3, M_3 = 1, M_4 = 0)$.

A state change is said to occur when there is a change in the values of any subset from the set of memory resources. One of the means of inducing or effecting a state change is through an event (see Section 2.1) or what is equivalent, a micro-operation. Note that is not the only agent of a state change. For example, in some microprogrammable machines, the contents of the control

memory address register is altered by a hardwired micro-sequencing unit. The state change effected in this case is not done through a micro-operation.

A state-based definition of parallelism in SLM's can now be given as follows:

Definition 4.1

Let S be an SLM and let $\mu_i < \mu_j$ in S. Then μ_i and μ_j are said to be locally parallel denoted $\mu_i \,||_L\, \mu_j$, if for all initial machine states, the execution of a micro-instruction $I = \{\mu_i, \mu_j\}$ produces the same final machine state as the sequential execution of $I_1=\{\mu_i\}$, $I_2 = \{\mu_j\}$.

Note that this definition merely makes more precise, the concept of parallelism as being able to "place a pair of MO's in the same microinstruction". The term "locally parallel" is used here to distinguish the parallelism within SLM's from "global" parallelism - which I shall discuss in Chapter V. The conditions for $\mu_i \,||_L\, \mu_j$ are of course, given by the expression (2.7), i.e.,

$$\mu_i \,||_L\, \mu_j \Longleftrightarrow (\mu_i \,\delta\, \mu_j) \vee (\mu_i \,\gamma\, \mu_j) \,. \qquad (4.1)$$

4.2.1 Significance of Branch Micro-operations

A micro-operation (MO) was defined in Section 2.1 as simply a control signal originating in the control memory which causes some event to take place. Here, I shall further distinguish between functional MO's (FMO)

and branch MO's (BMO).

BMO's represent conditional (two-way) or unconditional branches. In the context of the 5-tuple representation (2.4), it is assumed that SC denotes the set of arguments for the predicate defined by the (branch) OP, and SK designates the explicit destination of the BMO - the micro-operation to be executed next if the predicate is satisfied. For example, the notation

$$< \text{BHIGH}, \{R1,R2\}, \{\mu_k\}, \ U,V > \tag{4.2}$$

may mean that if R1 > R2, then control transfers to μ_k, else the next sequential micro-operation is accessed. Notice that the explicit destination in (4.2) is an MO only because the microprogram is specified in canonical form. In generating microinstructions, this explicit destination has to be transformed into a control memory word address, viz., the address of whichever microinstruction contains μ_k. The state change effected by the execution of a BMO therefore, is the assignment of a new value to the control memory address register only; no other memory resources are affected. An FMO is simply any MO other than a BMO.

Recall that in an SLM $S=<\mu_1,\mu_2,\ldots,\mu_t>$, the only entry and exit points are μ_1 and μ_t respectively. This implies that μ_t can be a BMO provided that the explicit destination of the branch is none of the MO's $\mu_2,\ldots,\mu_t$.

Furthermore if μ_t is a BMO then we have the following obvious property:

Lemma 4.1

Let $\mu_i < \mu_t$ for some pair of MO's μ_i, μ_t in an SLM such that μ_t is a BMO. If $I(\mu_i)$, $I(\mu_t)$ denote micro-instructions containing μ_i and μ_t respectively, then $I(\mu_t)$ cannot precede $I(\mu_i)$.

Proof

Since there is a single entry point (μ_1) and a single exit point (μ_t) in an SLM, evidently if any one MO is executed then so is every other MO in S. Let the execution of $I(\mu_t)$ precede that of $I(\mu_i)$; on executing $I(\mu_t)$ if the branch condition is satisfied, the next microinstruction to be executed is $I(\mu_e)$ where μ_e is the explicit destination of the BMO. In that case $I(\mu_i)$, and hence μ_i may be bypassed, contradicting the earlier assertion. □

The significance of this rather trivial lemma lies in that it serves to indicate that while designing a general optimizing algorithm, branch MO's must be treated as a special case.

4.2.2 Invertibility of Micro-operations

To motivate the approach developed below, consider the short example sequence of Fig. 4.1. We see that

$\sim(\mu_1 \,||_L\, \mu_2)$ and $\sim(\mu_2 \,||_L\, \mu_3)$, although the absence of parallelism in the two cases are for quite different reasons. If the Jackson-Dasgupta algorithm were to be applied to this example, we would obtain three microinstructions $I(\mu_6)$, $I(\mu_7)$, $I(\mu_8)$, and these would be executed in precisely this order.

Notice however, that μ_2 and μ_3 can be interchanged or <u>inverted</u> in the sequence without affecting the final result (machine state). If μ_2 and μ_3 <u>are</u> inverted, we obtain the (state) equivalent SLM $S_1^* = \langle\mu_1\mu_3\mu_2\rangle$ (Fig.4.2); and since $\mu_1 \,||_L\, \mu_3$, only two microinstructions are required, viz., $I(\mu_1,\mu_3)$ followed by $I(\mu_2)$. Since there are no other possible permutations we have in fact, obtained the minimal set of microinstructions.

μ_1 = < GATE, {1} , {2} ___ , Π1 >

μ_2 = < ADD, {2,3} , {4}, {ADDER}, Π1 >

μ_3 = < ADD, {3,5} , {5}, {ADDER}, Π1 >

<u>Fig. 4.1</u>

<u>An Example SLM : S_1</u>

μ_1 = < GATE, {1} , {2}, ___ , Π1 >

μ_3 = < ADD, {3,5} , {5}, {ADDER}, Π1 >

μ_2 = < ADD, {2,3} {4}, {ADDER}, Π1 >

<u>Fig. 4.2</u>

<u>S_1^* : An Inverted Version of S_1</u>

$\mu_4 : C \leftarrow A \wedge B;$

$\mu_5 : B \leftarrow D$

Fig. 4.3

An Example SLM : S_2

$\mu_5 : B \leftarrow D$

$\mu_4 : C \leftarrow A \wedge B$

Fig. 4.4

S_2^* : An Inverted Version of S_2

The reason that μ_2 and μ_3 can be inverted is of course, the fact that they employed disjoint sources and sinks. Ignoring for the present, the operational unit and time-validity components, consider the sequence S_2 (Fig. 4.3). The point is, can we legitimately invert these MO's?

Assuming that the memory elements are all 4-bit registers, and that states are represented as binary strings, suppose the initial states of A,B,D are respectively, "0000", "1101", and "1100". On executing S_2, the relevant final states are C = "0000" and B = "1100".

If this sequence is now inverted, S_2^* is obtained (Fig. 4.4). Given the same initial state notice that the final states are still C = "0000" and B = "1100", in spite of the use of the common resource B!

This is quite obviously due to the fact that A's initial state happened to be "0000". If it could be guaranteed that for <u>all</u> initial states, S_2 and S_2^* lead to identical final states then only would an <u>à priori</u> inversion of μ_4 and μ_5 be possible

In this analysis, it will be assumed that the microprogrammable processors are such that for any pair of MO's sharing data resources no such guarantee is possible. Or to state this more precisely, it is assumed that for any pair of MO's μ_i, μ_j that share data resources there exists at least one machine state ψ such that the execution of the sequences $<\mu_i \mu_j>$ and $<\mu_j \mu_i>$ with ψ as the initial state, lead to distinct final machine states. The notion of invertibility is then made precise by the following:

<u>Definition 4.2</u>

Let S be an SLM and μ_i, μ_j be in S. Then μ_i, μ_j are said to be <u>invertible</u>, denoted $\mu_i \lambda \mu_j$ if $\mu_i \beta \mu_j$.

One should note the distinction between the λ and β relations. The relation $\mu_i \beta \mu_j$ depends only on the details of the two MO's, whereas $\mu_i \lambda \mu_j$ depends in addition, on the appearance of μ_i and μ_j within an SLM. Like the β relation however, λ is symmetric.

Using Def. 4.2 in conjunction with the expression (4.1) leads to:

Theorem 4.1

Let S be an SLM, $\mu_i < \mu_j$ and $\sim(\mu_i \,||_L\, \mu_j)$. Then $\mu_i \lambda \mu_j$ if and only if

$$(V_i \cap V_j \neq \phi) \wedge (\mu_i \beta \mu_j) \wedge (U_i \cap U_j \neq \phi)$$

Proof

Assume that $\mu_i < \mu_j$, $\sim(\mu_i \,||_L\, \mu_j)$, and $\mu_i \lambda \mu_j$. Then $\mu_i \beta \mu_j$. Furthermore, under the above assuptions, $V_i \cap V_j \neq \phi$ must also be true. For otherwise, i.e. if $V_i \cap V_i = \phi$, then either $V_i < V_j$ or $V_i > V_j$ holds. But $V_i < V_j$ implies, by Definitions 2.1(iii), 2.2, and the expression (4.1) that $\mu_i \,||_L\, \mu_j$, a contradiction. Similarly $(V_i > V_j) \wedge (\mu_i \beta \mu_j)$ implies by Def. 2.1(ii) and the expression (4.1), that $\mu_i \,||_L\, \mu_j$, again a contradiction. Thus $V_i \cap V_j \neq \phi$. Assume now, that $U_i \cap U_j = \phi$. Then $(V_i \cap V_j \neq \phi) \wedge (\mu_i \beta \mu_j) \wedge (U_i \cap U_j = \phi)$ implies $\mu_i \,||_L\, \mu_j$, contradicting the assumption, hence $U_i \cap U_j \neq \phi$.

The converse is trivially true since $\mu_i < \mu_j$ and $\mu_i \beta \mu_j$ means, by definition, that $\mu_i \lambda \mu_j$. □

Suppose that in a given SLM, $\mu_i < \mu_j$ and $\mu_i \lambda \mu_j$; then the ordering $\mu_i < \mu_j$ is said to be the <u>specified</u> ordering. As a result of the λ relation, we may change the ordering from the specified one. The particular condition $\sim(\mu_i \,||_L\, \mu_j) \wedge (\mu_i \lambda \mu_j)$ will be denoted by the relation $\mu_i \lambda^* \mu_j$. That is, $\mu_i \lambda^* \mu_j$ represents the fact that a pair of non-parallel MO's may be inverted. For

example, referring to Fig. 4.1, $\mu_2 \lambda^* \mu_3$ is true.

Recall that if $\mu_i \,||_L\, \mu_j$, then μ_i, μ_j can be placed in a microinstruction, say I. Thus if $I = \{\mu_1, \mu_2, \ldots, \mu_k\}$, then for all pairs $\mu_i, \mu_j \in I$, $\mu_i \,||_L\, \mu_j$ - that is, I forms a parallel set. An ordering $<$ on a pair of microinstructions I_i, I_j, is defined such that if $I_i < I_j$ then I_i is executed before I_j. Thus, given an ordered sequence $I_1 < I_2 < \ldots < I_k$, it makes sense to speak of an "earlier" or "later" microinstruction. For convenience, I shall order microinstructions on their indices, i.e. $i < j$ implies $I_i < I_j$. Furthermore, as in Lemma 4.1, the notation $I(\mu_j)$ will denote a microinstruction containing μ_j. Finally, referring to Def. 2.1, the expressions (i), (ii) and (iii) of this definition will be distinguished by the relations $\mu_i \,\delta_1\, \mu_j$, $\mu_i \,\delta_2\, \mu_j$, and $\mu_i \,\delta_3\, \mu_j$ respectively.

Theorem 4.2

Let I_q be a microinstruction containing MO's from an SLM S, and μ_j an MO in S such that (i) $\mu_i < \mu_j$ in S for all $\mu_i \in I_q$; and (ii) μ_j is not already in I_q. Then

(a) If, for all $\mu_i \in I_q$, $(\mu_i \,\delta\, \mu_j \wedge SK_i \cap SK_j = \phi) \vee (\mu_i \lambda^* \mu_j)$, then some $I(\mu_j)$ can precede I_q.

(b) If there exists some $\mu_i \in I_q$ such that $\sim(\mu_i \,||_L\, \mu_j) \wedge \sim(\mu_i \,\lambda^*\, \mu_j)$ then I_q must precede $I(\mu_j)$.

(c) If there exists some $\mu_i \in I_q$ such that $\mu_i \,\gamma\, \mu_j$, and for all $\mu_k \in I_q - \{\mu_i\}$ $\mu_k \,||_L\, \mu_j$ then the earliest microinstruction for μ_j is I_q.

(d) If there exists some $\mu_i \varepsilon I_q$ such that $(\mu_i\ \delta_3\ \mu_j\ \Lambda$ $SK_i \cap SK_j \neq \phi)$, and for all $\mu_k \varepsilon I_q - \{\mu_i\}$, $\mu_k \, ||_L \, \mu_j$, then the earliest microinstruction for μ_j is I_q.

(e) If μ_j is a BMO then μ_j can be placed in I_q if and only if for all $\mu_i \varepsilon I_q$ $\mu_i \, ||_L \, \mu_j$ and there exists no other microinstruction I_n such that $I_q < I_n$.

Proof

(a) For some $\mu_i \varepsilon I_q$ if $\mu_i\ \lambda^*\ \mu_j$ then μ_j can always precede μ_i. On the other hand if $(\mu_i\ \delta\ \mu_j)\ \Lambda\ (SK_i \cap SK_j = \phi)$ then μ_i, μ_j are data independent since by Def. 2.1, $\mu_i\ \delta\ \mu_j$ implies. $\mu_i\ \alpha\ \mu_j$, and $(\mu_i\ \alpha\ \mu_j)\ \Lambda\ (SK_i \cap SK_j = \phi)$ means that $\mu_i\ \beta\ \mu_j$. Thus μ_i and μ_j can be placed in the same micro-instruction, or one can precede the other. If for all MO's in I_q one of the above conditions holds, then $I(\mu_j)$ can precede I_q, for some $I(\mu_j)$.

(b) $\sim(\mu_i \, ||_L \, \mu_j)$ implies that either $I(\mu_i) < I(\mu_j)$ or vice versa. But $\sim(\mu_i\ \lambda^*\ \mu_j)$ implies that the specified ordering must be preserved from which the statement follows.

(c) Let I_q be partitioned into $\{I_q', \mu_i\}$ such that, for all $\mu_k \varepsilon I_q'$ $\mu_k \, ||_L \, \mu_j$ and $\mu_i\ \gamma\ \mu_j$. Then μ_j can be placed in I_q. But $\mu_i\ \gamma\ \mu_j$ implies $\sim(\mu_i\ \alpha\ \mu_j)$, hence by Def. 4.2, the specified ordering $\mu_i < \mu_j$ cannot be changed. Thus $I(\mu_j)$ cannot precede $I_q = I(\mu_i)$.

(d) As above, partition I_q into $\{I_q', \mu_i\}$ such that for all $\mu_k \in I_q'$, $\mu_k \,||_L\, \mu_j$, and $(\mu_i\ \delta_3\ \mu_j) \wedge (SK_i \cap SK_j \neq \phi)$. Then obviously μ_j can be placed in I_q since I_q remains a parallel set of MO's. But $SK_i \cap SK_j \neq \phi$ implies $\sim(\mu_i\ \beta\ \mu_j)$ hence $\sim(\mu_i\ \lambda\ \mu_j)$, so that μ_j cannot precede μ_i. Thus $I(\mu_j)$ cannot precede $I(\mu_i)$.

(e) This statement follows trivially from the definition of a microinstruction and Lemma 4.1. $\square$

The reader will probably understand better, the above theorem, and its use in constructing the parallelism-detection algorithm (to be described below), with an example.

Consider an SLM $S = \langle \mu_1, \mu_2, \ldots, \mu_{10} \rangle$ from which a microinstruction $I_q = \{\mu_1, \ldots, \mu_5\}$ has already been constructed, where

$$
\begin{aligned}
\mu_1 &= \langle\ \text{GATE},\ \{R1\},\ \{A\},\ \_\_\_\ ,\ \Pi_1\ \rangle \\
\mu_2 &= \langle\ \text{GATE},\ \{R2\},\ \{B\},\ \_\_\_\ ,\ \Pi_1\ \rangle \\
\mu_3 &= \langle\ \text{ADD},\ \{A,B\},\ \{C\},\ \{\text{ADDER}\},\ \Pi_2\ \rangle \\
\mu_4 &= \langle\ \text{GATE},\ \{C\},\ \{R3\},\ \_\_\_\ ,\ \Pi_3\ \rangle \\
\mu_5 &= \langle\ \text{GATE},\ \{C\},\ \{R4\},\ \_\_\_\ ,\ \Pi_3\ \rangle
\end{aligned}
\tag{4.3}
$$

and $\Pi_1 < \Pi_2 < \Pi_3$. The remaining MO's in S are given by

$$\begin{array}{llllll} \mu_6 = < & \text{GATE}, & \{R1\}, & \{E\}, & \_\_\_, & \Pi_1 > \\ \mu_7 = < & \text{GATE}, & \{R3\}, & \{A\}, & \_\_\_, & \Pi_1 > \\ \mu_8 = < & \text{SHL}, & \{A\} & \{D\} & ,\{\text{SHIFTER}\}, & \Pi_2 > \\ \mu_9 = < & \text{GATE}, & \{R3\}, & \{B\}, & \_\_\_, & \Pi_2 > \\ \mu_{10} = < & \text{BHIGH}, & \{R3,R5\}, & \{\text{"}\mu_k\text{"}\}, & \{\text{MSEQR}\}, & \Pi_1 > \end{array} \tag{4.4}$$

We may then make the following observations:

[1] For all $\mu_i \in I_q$, $\mu_i \,\delta\, \mu_6$ and $SK_i \cap SK_6 = \phi$; hence by statement (a) of Theorem 4.2, μ_6 can precede I_q.

[2] μ_7 cannot precede I_q since $\sim(\mu_4 \;\lambda^*\; \mu_7)$; also, μ_7 cannot be placed in I_q since $\sim(\mu_4 \;||_L\; \mu_7)$; hence, by statement (b) $I_q < I(\mu_7)$.

[3] Since $\mu_1 \,\gamma\, \mu_8$, and for all other $\mu_i \in I_q, \mu_i \;||_L\; \mu_8$, μ_8 can be placed in I_q. But since $\sim(\mu_1 \,\alpha\, \mu_8)$, μ_8 cannot precede I_q. Thus, by statement (c), the earliest microinstruction for μ_8 is I_q.

[4] Since $\mu_2 \;\delta_3\; \mu_9$, and for all other $\mu_i \in I_q$, $\mu_i \;||_L\; \mu_9$, μ_9 can be placed in I_q. But again, μ_9 cannot precede I_q since $\sim(\mu_2 \,\beta\, \mu_9)$. Hence by statement (d), the earliest microinstruction for μ_9 is I_q.

[5] Finally, note that since $\sim(\mu_4 \;||_L\; \mu_{10})$, and μ_{10} is a BMO, I_q must precede $I(\mu_{10})$ - from statement (e).

4.3 The Optimizing Algorithm

The algorithm can now be presented. This algorithm uses three pointer variables as follows: "i" is a pointer

to the microinstruction "currently" being examined; "i*" points to the latest microinstruction in the ordered sequence of microinstructions generated at any given time; and "j" points to an element of the input SLM. The expression "branch (n)" denotes a predicate which is TRUE if n is a branch micro-operation, and is FALSE otherwise.

<u>Algorithm 4.1</u>: Detection of Parallel Micro-operations in an SLM.

<u>Input</u>: An SLM $S = <\mu_1, \mu_2, \ldots, \mu_t>$.

<u>Output</u>: An ordered sequence of microinstructions

$I = <I_1 I_2, \ldots, I_r>, \quad r \leq t$.

[1] $i \leftarrow i^* \leftarrow 1;\ j \leftarrow 0;$

[2] $I_1 \leftarrow \{\mu_1\};$

[3] $j \leftarrow j+1;\ a \leftarrow 0;$

<u>If</u> $j > t$ <u>then</u> $I \leftarrow \{I_1, I_2, \ldots, I_{i^*}\}$; STOP.

[4] <u>If</u> branch (μ_j) <u>then</u>

<u>begin</u>

<u>if</u> $\mu ||_L \mu_j \;\forall\; \mu \in I_i$

[4a] <u>then</u> $I_i \leftarrow I_i \cup \{\mu_j\}$

<u>else</u>

<u>begin</u>

[4b] $I_{i+1} \leftarrow \{\mu_j\};\ i \leftarrow i+1;\ i^* \leftarrow i$

<u>end</u>

<u>goto</u> [3]

<u>end</u>

[5] **If** $\exists\, \mu \in I_i \ni \sim(\mu \,||_L\, \mu_j) \wedge \sim(\mu\, \lambda^*\, \mu_j)$

  then

    begin

      $i \leftarrow i + 1;\; i^* \leftarrow i;$

      $I_i \leftarrow \{\mu_j\};$

      goto [3]

    end

[6] **If** $(\exists\, \mu \in I_i \ni \mu\, \gamma\, \mu_j) \wedge (\mu' \,||_L\, \mu_j \;\forall\; \mu' \in I_i - \{\mu\})$

  then

    begin

      $I_i \leftarrow I_i \cup \{\mu_j\};$

      goto [3]

    end

[7] **If** $(\exists\, \mu \in I_i \ni \mu\, \delta\, \mu_j \wedge SK \cap SK_j \neq \phi) \wedge (\mu' \,||_L\, \mu_j \;\forall\; \mu' \in I_i - \{\mu\})$

  then

    begin

      $I_i \leftarrow I_i \cup \{\mu_j\};$

      goto [3]

    end

[8] **While** $[(\mu\, \delta\, \mu_j \wedge SK \cap SK_j = \phi) \vee (\mu\, \lambda^*\, \mu_j) \;\forall\; \mu \in I_i] \wedge [i > 0]$

  do **begin**

      if $(\mu\, \delta\, \mu_j \wedge SK \cap SK_j = \phi) \;\forall\; \mu \in I_i$

        then $a \leftarrow i;$

      $i \leftarrow i - 1$

    end

```
[9]     If i = 0 then
          begin
[9a]        if a ≠ 0 then I_a ← I_a ∪ {μ_j}
              else
[9b]            begin
                  k ← i*;
                  while k > 0 do
                    begin
                      I_k+1 ← I_k;
                      k ← k - 1
                    end
                  I_k+1 ← {μ_j};
                  i* ← i* + 1
                end
[9c]        i ← i*;
            goto [3]
          end

[10]    While ∃ μ ε I_i ∋ ∼(μ ||_L μ_j) do
          begin
            i ← i + 1;
            if i > i' then
                        begin
[10a]                     I_i ← {μ_j}; i* ← i;
                          goto [3]
                        end
          end
```

[11] $I_i \leftarrow I_i \cup \{\mu_j\}$; $i \leftarrow i^*$;
 goto [3] . □

Verification of the algorithm proceeds by induction on L(S), the length of the input SLM S. I shall first show that for any partition I_g in the output set I, $\mu_j, \mu_k \in I_g$ satisfy $\mu_j \; ||_L \; \mu_k$. That is, each I_g obtained is indeed a microinstruction. I shall also show that the output satisfies the necessary precedence constraints imposed by Theorem 4.2.

Theorem 4.3

Let $I = \{I_1, I_2, \ldots, I_i\}$ be the output produced by Algorithm 4.1. Then

(a) For all $\mu_j, \mu_k \in I_g$ in I, $\mu_j \; ||_L \; \mu_k$;

(b) If $\mu_j < \mu_k$ in S, and $\sim(\mu_j \; ||_L \; \mu_k) \wedge \sim(\mu_j \; \lambda^* \; \mu_k)$ then $I(\mu_j) < I(\mu_k)$ in I.

Proof

First note that at the start of each iteration (i.e., whenever Step [3] is entered), $i = i^*$ denotes the index of the "latest" partition generated. This itself can be proved by induction on the number of times Step [3] is entered. For, it is certainly true the first time the step is entered since by Steps [1], [2], $i = i^* = 1$, and only one microinstruction I_1 exists.

Assume that this is true just before the m-th iteration of Step [3], and let $i = i^* = n$ at this stage. Then

(i) the only steps in which i is incremented, are Steps [4b], [5], or [10a], and in each of these steps, a "latest" microinstruction $I_i = I_{n+1}$ is created and i* made equal to i; (ii) in Steps [4a], [6], [7] or [11], $I_i = I_n$ remains the latest microinstruction and i* remains unchanged at n; and (iii) in Step [9], either i and i* remain unchanged at the value n, and no new microinstruction is created (Steps [9a,9c]), or a new "latest" microinstruction I_{n+1} is constructed and i,i* both made equal to n+1 (Steps [9b,9c]). Thus, at the beginning of the (m+1)-th iteration of Step [3], i = i* denotes the index of the latest partition constructed thus far.

To prove the two statements of the above theorem, denote by L(S), the length of the input, S. For L(S) = 2, $S = \langle\mu_1\mu_2\rangle$ (say). Then, by Step [2], $I_1 = \{\mu_1\}$. The only steps by which μ_2 is placed in I_1, are [4a], [6], [7] and [11], and in all these cases, $\mu_1 \,||_L\, \mu_2$ is satisfied. Hence, statement (a) of Theorem 4.3 is proved. If however, $\sim(\mu_1 \,||_L\, \mu_2) \wedge \sim(\mu_1 \;\lambda^*\; \mu_2)$ then either Steps [4b] or [5] is entered, and in either of these, μ_2 is placed in I_2. This proves statement (b).

Suppose as the induction hypothesis, that the theorem is true for a length n-1, and consider μ_n, the n-th MO. Without loss of generality, denote the current set of partitions by

$$I^* = \{I_1, I_2, \ldots, I_i\} \; .$$

It is easy to see that proposition (a) holds since the only conditions under which μ_n is placed in an existing partition I_k are when for all $\mu \varepsilon I_k$, $\mu ||_L \mu_n$ (Steps [4a], [6], [7], [9a], [11]). In the remaining cases, a new partition is created for μ_n (Steps [4b], [5], [9b], [10]). Thus, in the case of an existing partition I_k, all the MO's remain pairwise parallel.

Consider the second proposition. If μ_n is a BMO, then by Step [4], μ_n is placed either in I_i or in I_{i+1}. If μ_n is put in I_i then any μ_j in S satisfying $(\mu_j < \mu_n) \wedge \sim(\mu_j ||_L \mu_n)$ will not be in I_i, hence $I(\mu_j) < I(\mu_n)$ since I_i is the latest microinstruction. If there does exist a μ_j in I_i such that $(\mu_j < \mu_n) \wedge \sim(\mu_j ||_L \mu_n)$ then μ_n will be placed in I_{i+1}, i.e., $I(\mu_j) < I(\mu_n)$ since $I_i < I_{i+1}$ by assumption. In either case then, statement (b) is satisfied since, if μ_n is a BMO and $\mu_j < \mu_n$ in S, then $\sim(\mu_j \lambda \mu_n)$ implicitly holds.

Let μ_n be an FMO, and let the condition of Step [5] be satisfied. Then μ_n is placed in a new partition I_{i+1}, and $I(\mu_j) < I(\mu_n)$ for all $\mu_j < \mu_n$ in S.

If the condition of Step [5] is not satisfied, then for all $\mu \varepsilon I_i$, $\mu ||_L \mu_n$ or $\mu \lambda^* \mu_n$. If now, the condition of Step [6] is satisfied, then for all $\mu \varepsilon I_i$, $\mu ||_L \mu_n$, and $I(\mu_n) = I_i$. If there exists any $\mu_j < \mu_n$ in S such that $\sim(\mu ||_L \mu_n) \wedge \sim(\mu_j \lambda^* \mu_n)$ then $\mu_j \not\varepsilon I_i$. Thus $I(\mu_j) < I(\mu_n)$.

If the condition of Step [6] does not hold, then $(\mu\ \lambda^*\ \mu_n) \vee (\mu\ \delta\ \mu_n)$ holds for all $\mu \in I_i$. If the condition of Step [7] is now satisfied, $I(\mu_n) = I_i$, and as above $I(\mu_j) < I(\mu_n)$. Otherwise the next step, [8] is entered in which case the condition $(\mu\ \lambda^*\ \mu_n) \vee [(\mu\ \delta\ \mu_n) \wedge (SK \cap SK_n = \phi)]$ is true. An exit from Step [8] is obtained when either of the following conditions is satisfied:

[a] $(i = 0) \wedge [(\mu\ \delta\ \mu_n \wedge SK \cap SK_n = \phi) \vee (\mu\ \lambda^*\ \mu_n)$ for all $\mu \in I_p$, for all I_p, $p = 1,\ldots,i^*]$.

[b] $i = h$ for some h satisfying $1 \leq h \leq i^* - 1$ such that $[\sim(\mu\ \delta\ \mu_n) \vee (\mu\ \delta\ \mu_n \wedge SK \cap SK_n \neq \phi)] \wedge [\sim(\mu\ \lambda^*\ \mu_n)]$ for some $\mu \in I_h$.

Condition [a] leads to two possibilities, viz:

[a1] For all I_p $(p = 1,\ldots,i^*)$ and for all $\mu \in I_p$, $\mu\ \lambda^*\ \mu_n$ is true. In that case $a = 0$, so that by Step [9b] μ_n is placed alone in I_1. Moreover, since condition [a] is satisfied, there exists no $\mu_j < \mu_n$ in S such that $\sim(\mu_j\ ||_L\ \mu_n) \wedge \sim(\mu_j\ \lambda^*\ \mu_n)$ holds, so that placing μ_n in I_1 does not violate statement (b) of the Theorem.

[a2] There exists some I_p $(1 \leq p \leq i^*)$ such that, for all $\mu \in I_p$, $\sim(\mu\ \lambda^*\ \mu_n)$, but $(\mu\ \delta\ \mu_n \wedge SK \cap SK_n = \phi)$ is true. By Step [8], the variable "a" points to the "earliest" microinstruction satisfying this condition. Since $a \neq 0$,

μ_n is placed in I_a by Step [9a]. Furthermore, since condition [a] above is still satisfied, we see that proposition (b) of the theorem is not violated.

Under the condition [b] above, since $i \neq 0$, Step [10] is executed. Note that this condition means either

(i) $\sim(\mu\ \delta\ \mu_n) \wedge \sim(\mu\ \lambda^*\ \mu_n)$ for some $\mu \in I_h$; or

(ii) $(\mu\ \delta\ \mu_n) \wedge (SK \cap SK_n \neq \phi) \wedge \sim(\mu\ \lambda^*\ \mu_n)$.

In the case of (i), though $\sim(\mu\ \delta\ \mu_n)$, it is possible that $\mu\ \gamma\ \mu_n$ in which case, although $\mu \,||_L\, \mu_n$ is true, so is $\sim(\mu\ \lambda\ \mu_n)$. If neither $\mu\ \delta\ \mu_n$ nor $\mu\ \gamma\ \mu_n$ are true, then $\sim(\mu \,||_L\, \mu_n)$, and since $\sim(\mu\ \lambda^*\ \mu_n)$, this implies $\sim(\mu\ \lambda\ \mu_n)$. So in any case, I_h is the earliest possible microinstruction for μ_n.

Similarly, if (ii) holds, $SK \cap SK_n \neq \phi$ imply $\sim(\mu\ \lambda\ \mu_n)$. Again, the earliest possible microinstruction for μ_n is I_h.

Thus, if there exists some $\mu_j < \mu_n$ in S such that $\sim(\mu_j \,||_L\, \mu_n) \wedge \sim(\mu_j\ \lambda^*\ \mu_n)$ then $I(\mu_j) \leq I_h$ since otherwise the earliest possible microinstruction would have been some $I_{h*} > I_h$. If $I(\mu_j) = I_h$, Step [10] ensures that μ_n is placed in a later microinstruction so that $I(\mu_j) < I(\mu_n)$. If $I(\mu_j) < I_h$, then by Step [10], $I(\mu_n) = I_h$, so that again, $I(\mu_j) < I(\mu_n)$, thereby satisfying proposition (b) of the theorem. □

The second part of the verification is concerned with the minimality of the output.

Theorem 4.4

For any input SLM S, let $I = \{I_1, I_2, \ldots, I_r\}$ be the output produced by Algorithm 4.1. Then I is such that there exists in I_j $(2 \leq j \leq r)$ at least one MO which cannot be placed in an earlier microinstruction.

Proof

By induction on the length L(S) of the input; for L(S) = 1,2, the proof is trivial. Suppose the theorem is true for SLM's of length n-1, and let the output be denoted by

$$I = \{I_1, I_2, \ldots, I_r\} .$$

The assumption of minimality means that there exists in $\langle I_i, I_{i+1} \rangle$, $i = 1,2,\ldots,r-1$, at least one pair of micro-operations $\mu^i \varepsilon I_i$, $\mu^{i+1} \varepsilon I_{i+1}$ such that the execution of μ^i and μ^{i+1} can never take place in the same micro-instruction; and that I is the smallest set of micro-instructions satisfying all precedence requirements.

Considering the n-th MO μ_n, we can immediately see that the cardinality $|I|$ of I can never be made less than r because of the induction hypothesis. Thus the minimum possible value of $|I|$ is either r or r+1.

In the special case where μ_n is a BMO, then by Theorem 4.2, $|I| = r$ if all $\mu \varepsilon I_r$, $\mu ||_L \mu_n$; otherwise

$|I| = r+1$. These are the minimal possible values of $|I|$. That these values are indeed obtained by Step [4] is easily seen. Suppose μ_n is not a BMO. Then the only steps where an additional microinstruction is created are [5], [9b] and [10]. In all other cases μ_n is placed in an existing microinstruction so that $|I|$ remains r. It is thus sufficient to show that under the conditions leading to additional microinstructions, μ_n <u>must</u> be placed in a new partition.

[1] The condition of Step [5] requires (by Theorem 4.2(b)) that $I_r < I(\mu_n)$ hence μ_n must be placed in I_{r+1}.

[2] In Step [9b], condition [a1] in the proof of Theorem 4.3 is satisfied; i.e., for all I_i in I, $\mu \lambda^* \mu_n$ for all μ in I_i. Hence μ_n cannot be placed in an existing microinstruction, so a new partition must be created for μ_n.

[3] In Step [10], if the condition "$\exists\, \mu \in I_i \ni \sim(\mu ||_L \mu_n)$" is satisfied then μ_n cannot be placed in I_i. If this condition is satisfied for <u>all</u> $I_i \in I$, μ_n has to be placed (as indeed it is by the algorithm) in a new partition so that $|I| = r+1$. This completes the proof of the theorem.

4.3.1 An Example

To demonstrate the application of the algorithm, I shall use the hypothetical SLM specified below.

$$
\begin{array}{lllllll}
\mu_1 = & \langle \text{GATE}, & \{\text{R1}\}, & \{\text{A}\}, & \underline{\qquad}, & \Pi_1 & \rangle \\
\mu_2 = & \langle \text{GATE}, & \{\text{R2}\}, & \{\text{B}\}, & \underline{\qquad}, & \Pi_1 & \rangle \\
\mu_3 = & \langle \text{ADD}, & \{\text{A,B}\}, & \{\text{C}\}, & \{\text{ADDER}\}, & \Pi_2 & \rangle \\
\mu_4 = & \langle \text{GATE}, & \{\text{C}\}, & \{\text{R3}\}, & \underline{\qquad}, & \Pi_2 & \rangle \\
\mu_5 = & \langle \text{GATE}, & \{\text{C}\}, & \{\text{R4}\}, & \underline{\qquad}, & \Pi_2 & \rangle \\
\mu_6 = & \langle \text{INCR}, & \{\text{R1}\}, & \{\text{R1}\}, & \{\text{INCR}\}, & \Pi_2 & \rangle \\
\mu_7 = & \langle \text{GATE}, & \{\text{R1}\}, & \{\text{MAR}\}, & \underline{\qquad}, & \Pi_1 & \rangle \\
\mu_8 = & \langle \text{AND}, & \{\text{R3,R8}\}, & \{\text{A}\}, & \{\text{LOGIC}\}, & \Pi_2 & \rangle \\
\mu_9 = & \langle \text{NOT}, & \{\text{R2}\}, & \{\text{R6}\}, & \{\text{LOGIC}\}, & \Pi_2 & \rangle
\end{array}
$$

Fig. 4.5

Hypothetical Input to Algorithm 4.1 with $\Pi_1 < \Pi_2$

Construction of the output set of microinstructions by the algorithm, is demonstrated by the sequence of partition sets that is progressively obtained (Fig. 4.6). In contrast, consider the application of the non-optimizing JD algorithm to the same example. The microinstruction set obtained in this case is given as Fig. 4.7. Note that an additional microinstruction is required.

By Step [2] : $I_1 = \{\mu_1\};$

By Step [9a] : $I_1 = \{\mu_1, \mu_2\};$

By Step [6] : $I_1 = \{\mu_1, \mu_2, \mu_3\};$

By Step [5] : $I_1 = \{\mu_1, \mu_2, \mu_3\};$ $I_2 = \{\mu_4\};$

By Step [11] : $I_1 = \{\mu_1, \mu_2, \mu_3\};$ $I_2 = \{\mu_4, \mu_5\};$

By Step [11] : $I_1 = \{\mu_1, \mu_2, \mu_3, \mu_6\};$ $I_2 = \{\mu_4, \mu_5\};$

By Step [11] : $I_1 = \{\mu_1, \mu_2, \mu_3, \mu_6\};$ $I_2 = \{\mu_4, \mu_5, \mu_7\};$

By Step [5] : $I_1 = \{\mu_1, \mu_2, \mu_3, \mu_6\};$ $I_2 = \{\mu_4, \mu_5, \mu_7\};$
$I_3 = \{\mu_8\};$

By Step [11] : $I_1 = \{\mu_1, \mu_2, \mu_3, \mu_6, \mu_9\};$ $I_2 = \{\mu_4, \mu_5, \mu_7\};$
$I_3 = \{\mu_8\}.$

Fig. 4.6

Construction of the Microinstruction Set for the Example of Fig. 4.5

$I_1 = \{\mu_1, \mu_2, \mu_3, \mu_6\};$

$I_2 = \{\mu_4, \mu_5, \mu_7\};$

$I_3 = \{\mu_8\};$

$I_4 = \{\mu_9\};$

Fig. 4.7

Output of the JD Algorithm for the Input of Fig. 4.5

4.4 Conclusions

As I have remarked earlier, Algorithm 4.1 is of somewhat greater generality than the optimizing algorithms of Tsuchiya and Gonzalez [72] or Yau *et al* [77], since it is applicable to the more problematic case of polyphase microprograms. The proposed algorithm is then essentially an optimizing version of the Jackson-Dasgupta algorithm [39].

Consider now, the computational (time) complexity of Algorithm 4.1. Using the number of comparisons between pairs of micro-operations as a measure of this complexity, we obtain the following result:

Theorem 4.5

Algorithm 4.1 requires $0(n^2)$ comparisons where n is the size of the input SLM.

Proof

Consider the k-th MO μ_k for $2 \leq k \leq n$. For μ_k, one and only one of the following step sequences will be executed: [4]; [5]; [6]; [7]; [8], [9]; [8], [10]; or [8], [10], [11]. It is easily seen that the longest case corresponds to the step sequence [8], [10] since complete backtracking may be involved here.

Suppose the partitions already obtained are:

$$I_1, I_2, \ldots, I_{i^*}$$

$$\longleftarrow$$

with the arrow indicating the "direction" of comparisons in Step [8]. There are k-1 MO's in these partitions. In the worst case then, on exiting from Step [8], i = 1, so that μ_k has already been compared with these k-1 MO's. In Step [10], the "direction" of comparisons is reversed:

$$I_1, I_2, \ldots, I_{i^*}$$
$$\xrightarrow{\hspace{2cm}}$$

The worst case will then occur if μ_k cannot be placed in any of the existing partitions, in which case Step [10] causes μ_k to be compared a further (k-1) times while backtracking.

In the worst case then, μ_k requires a total of 2(k-1) comparisons, and if this happens for each of the MO's $\mu_2, \mu_3, \ldots, \mu_n$, (the worst possible case), the total number of comparisons is

$$\sum_{k=2}^{n} 2(k-1) .$$

Thus, the time complexity using this particular measure is $O(n^2)$. □

Complexity-wise then, Algorithm 4.1 is of the same order as the JD algorithm since the latter requires $O(n^2)$ comparisons of MO pairs in order to construct the conflict graph.

CHAPTER V

PARALLELISM IN LOOP-FREE MICROPROGRAMS

5.1 Introduction

Consider the canonical microprogram shown in Fig. 5.1. If Algorithm 4.1 is applied separately to each of the straight-line segments (demarcated here by dashed lines), then a total of 9 microinstructions is obtained (Fig. 5.2). However, one may easily observe that μ_9 can be executed along with μ_1 and μ_2, and μ_{10} with μ_3 without changing the final machine state; by doing so, the resulting number of microinstructions reduces to 7 (Fig. 5.3).

This example illustrates how a more "global" analysis of the input canonical microprogram may often yield a better output than (local) analysis of the straight-line components alone.

I shall refer to the phenomenon wherein MO's not necessarily belonging to the same SLM are placeable in the same microinstruction as global parallelism. Thus parallelism within an SLM is a special (local) case of global parallelism. The aim of the present chapter is to develop a partial theory of global parallelism in microprograms, and by applying this theory, extend Algorithm 4.1 to the more general case of loop-free

microprograms. The basic idea behind the analysis is the concept of <u>code motion</u> as illustrated in the above example.

μ_1	: < GATE,	{R1} ,	{AIL} ,	___ ,	V_1	>
μ_2	: < SHL,	{AIL} ,	{AO} ,	{ALU} ,	V_2	>
μ_3	: < GATE,	{AO} ,	{R1} ,	___ ,	V_1	>
μ_4	: < BZ,	{R1} ,	{'μ_9'} ,	{SEQR} ,	V_1	>
μ_5	: < GATE,	{R1}	{AIL} ,	___ ,	V_1	>
μ_6	: < GATE,	{R3}	{AIR} ,	___ ,	V_1	>
μ_7	: < ADD,	{AIL,AIR},	{AO} ,	{ALU} ,	V_2	>
μ_8	: < GATE,	{AO} ,	{R2} ,	___ ,	V_1	>
μ_9	: < GATE,	{R4} ,	{MAR} ,	___ ,	V_1	>
μ_{10}	: < READMEM,	{MEM} ,	{MBR} ,	___ ,	V	>
μ_{11}	: < GATE,	{MBR} ,	{R1} ,	___ ,	V_1	>
μ_{12}	: < GATE,	{R1} ,	{AIL} ,	___ ,	V_1	>
μ_{13}	: < GATE,	{R2} ,	{AIR} ,	___ ,	V_1	>
μ_{14}	: < ADD,	{AIL,AIR},	{AO} ,	{ALU} ,	V_2	>

$$(V = \langle V_1, V_2 \rangle)$$

<u>Fig. 5.1</u>

<u>Loop-Free Canonical Microprogram</u>

$I_1 = \{\mu_1, \mu_2\}$

$I_2 = \{\mu_3\}$

$I_3 = \{\mu_4\}$

$I_4 = \{\mu_5, \mu_6, \mu_7\}$

$I_5 = \{\mu_8\}$

$I_6 = \{\mu_9, \mu_{13}\}$

$I_7 = \{\mu_{10}\}$

$I_8 = \{\mu_{11}\}$

$I_9 = \{\mu_{12}, \mu_{14}\}$

Fig. 5.2

Output from Algorithm 4.1

$I_1 = \{\mu_1, \mu_2, \mu_9\}$

$I_2 = \{\mu_3, \mu_{10}\}$

$I_3 = \{\mu_4\}$

$I_4 = \{\mu_5, \mu_6, \mu_7\}$

$I_5 = \{\mu_8\}$

$I_6 = \{\mu_{11}\}$

$I_7 = \{\mu_{12}, \mu_{13}, \mu_{14}\}$

Fig. 5.3

"Globally" Analysed Output

The use of code motion transformation in program optimization is well known [4,5]. The usual objective there is to remove some invariant piece of code from within a loop so as to reduce the number of times that the code segment is executed. In the present context, code motion will be utilised only to enable (if possible) better compaction of MO's, i.e. to generate as few micro-instructions as possible.

One must note however, that analysing a micro-program for global parallelism may often prove to be fruitless: the resulting output may be as large as is

produced by purely local analysis. Indeed, such improvements as demonstrated by the example of Fig. 5.1 would have been unnecessary had the original microprogram been manually optimized by (the programmer) noticing for instance that μ_9, μ_{10} could have been part of the first rather than the last SLM.

Against this observation I offer the argument that the whole objective of automatic optimization is to permit the programmer to concentrate on the problem of microprogram correctness rather than on efficiency. If the code segment of Fig. 5.1 is "correct", the microprogrammer's task is done. It is up to the optimizer (or more generally, the compiler) to transform and if possible, improve the code.

Global analysis then, offers a possible strategy for microprogram optimization. In some cases it will yield better code than can be produced by purely local analysis (as in the case of Fig. 5.1); in other cases there will be no improvement, as for instance, for the segment shown in Fig. 5.4. The choice of using or rejecting global analysis as a means of optimization is an implementation decision. I should point out however, that the algorithms presented in this chapter are such that the output produced will certainly be no worse than that produced by local analysis. Hence the real price to be paid is greater compilation/optimization time.

This aspect will be discussed further below.

$$\mu_1 : < \text{GATE} \quad \{R4\} \quad \{MAR\} \quad \_\_\_ , V_1 >$$

$$\mu_2 : < \text{READMEN}, \{MEM\} , \{MBR\} , \_\_\_ , V >$$

$$\mu_3 : < \text{GATE}, \{R1\} , \{AIL\} , \_\_\_ , V_1 >$$

$$\mu_4 : < \text{SHL}, \{AIL\} , \{AO\} , \{ALU\} , V_2 >$$

$$\mu_5 : < \text{GATE}, \{AO\} , \{R1\} , \_\_\_ , V_1 >$$

$$\mu_6 : < \text{BZ}, \{R1\} , \{"\mu_{11}"\} , \_\_\_ , V_1 >$$

- - - - - - - - - - - - - - - - - - - -

$$\mu_7 : < \text{GATE}, \{R1\} , \{AIL\} , \_\_\_ , V_1 >$$

$$\mu_8 : < \text{GATE}, \{R3\} , \{AIR\} , \_\_\_ , V_1 >$$

$$\mu_9 : < \text{ADD}, \{AIL,AIR\}, \{AO\} , \{ALU\} , V_2 >$$

$$\mu_{10} : < \text{GATE}, \{AO\} , \{R2\} , \_\_\_ , V_1 >$$

- - - - - - - - - - - - - - - - - - - -

$$\mu_{11} : < \text{GATE}, \{MBR\} , \{R1\} , \_\_\_ , V_1 >$$

$$\mu_{12} : < \text{GATE}, \{R1\} , \{AIL\} , \_\_\_ , V_1 >$$

$$\mu_{13} : < \text{GATE}, \{R2\} , \{AIR\} , \_\_\_ , V_1 >$$

$$\mu_{14} : < \text{ADD}, \{AIL,AIR\}, \{AO\} , \{ALU\} , V_2 >$$

Fig. 5.4

Manually Transformed Version of Microcode Segment in Fig. 5.1

The general problem of detecting parallel tasks in branch containing task streams, have been studied previously by other authors [44,69]. Kuck *et al* [44] were

concerned with the analysis of FORTRAN-type statements including DO-loops. Tjaden and Flynn [69] used transition matrices for the dynamic detection of concurrency in instruction streams. They pointed out that even if conditional branches are present in the instruction stream, it is still possible to identify segments which would always execute regardless of the branch decision. This particular concept forms the basis for the present analysis.

5.2 Microprogram Flowgraphs and Symmetric Pairs

A canonical microprogram S can be transformed into a set of SLM's together with a specification of the precedence relationships between the SLM's, using the method proposed by Ramamoorthy and Gonzales [53]. More precisely, it is assumed that the canonical microprogram is in the form of a flowgraph defined as follows [2]:

Definition 5.1

A flowgraph is a labelled, directed graph G, containing a distinguished vertex v such that every vertex in G is reachable from v. Vertex v is called the begin vertex.

Definition 5.2

A flowgraph of a canonical microprogram S, is a flowgraph G_S in which each vertex corresponds to an SLM.

Let each vertex be labelled by the name of the SLM it represents. Then an edge e_{ij} is drawn from vertex S_i to vertex S_j if

(i) the last MO in S_i is neither a BMO nor a HALT, and S_j follows S_i in S; or

(ii) the last MO in S_i is a BMO, and the first MO in S_j is either an explicit or implicit destination of the BMO.

Figs. 5.5 and 5.6 schematize two canonical microprograms. The corresponding flowgraphs are given by Figs. 5.7 and 5.8 respectively.

Consider the execution of the microprogram represented by Figs. 5.5 and 5.7. Clearly, regardless of the branch decision at μ_4, S_1 and S_3 will always be executed. Furthermore, they will be executed exactly once. On the other hand, depending on the decision at μ_4, S_2 may not be executed at all, or it may be executed several times. Similarly, in Fig. 5.8 S_6 is executed if and only if S_1 is executed, and S_5 is executed if and only if S_2 is executed.

Thus, given an arbitrary flowgraph G_s, we may identify pairs of vertices S_i, S_j satisfying the property that S_i is executed if and only if S_j is executed. Such vertex pairs will be called <u>symmetric pairs</u>. Their significance lies in that MO's in symmetric pairs are potential candidates for global parallelism. Note in

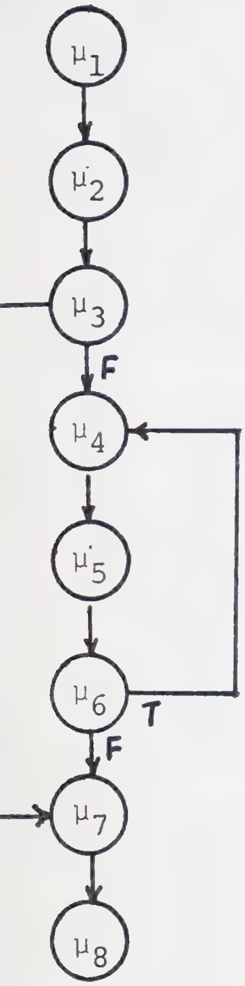

Fig. 5.5

nonical Microprogram
ontaining a Loop

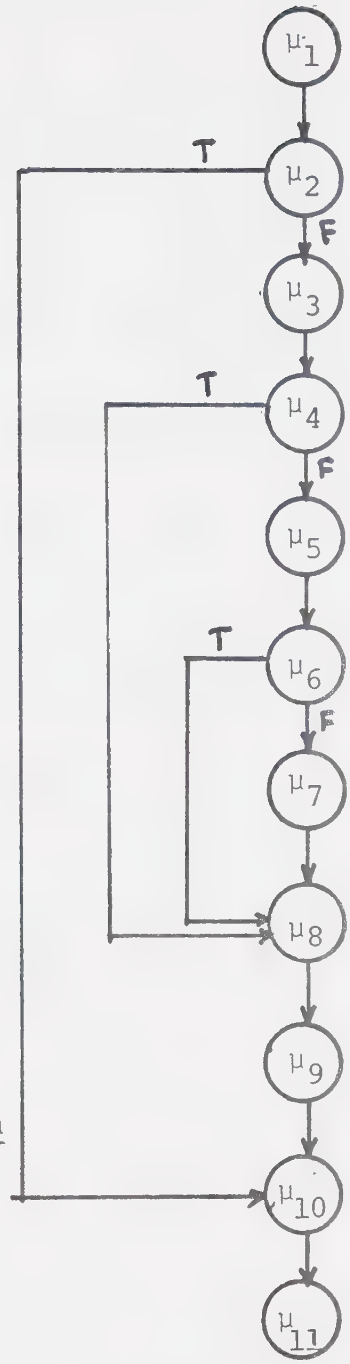

Fig. 5.6

Canonical Microprogram
Without a Loop

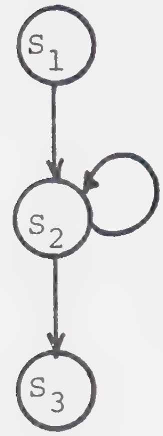

Fig. 5.7

Flowgraph of Fig.5.5

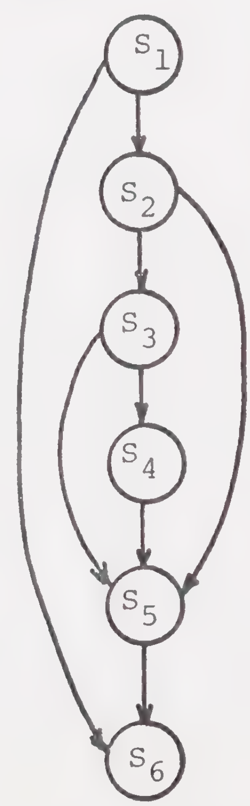

Fig. 5.8

Flowgraph of Fig. 5.6

Fig. 5.8 that S_3 and S_5 do not form a symmetric pair since the execution of S_5 does not imply the execution of S_3.

Consider a directed path

$$P = S_1 e_{12} S_2 e_{23} \cdots S_{k-1} e_{k-1,k} S_k \qquad (5.1)$$

within a flowgraph, where the S_i's and e_{jk}'s denote respectively, the vertices and edges in the path. Then P is said to include $S_1, S_2, \ldots, S_k$; also, P is said to be from S_1 to S_k. If invalence $(S_1) = 0$ and outvalence $(S_k) = 0$, the path P is said to be maximal. Intuitively, a directed path P is maximal if it cannot be extended by an edge at either end. Given a maximal directed path P from S_1 to S_k, S_1 is said to be the origin and S_k the terminus of P.

A path P_i in G_s is distinct if there exists no other path P_j in G_s such that $E(P_i) = E(P_j)$ where $E(p)$ denotes the edge set in P.

Note that a directed path in a flowgraph may include a directed circuit as a subpath. For instance, Fig. 5.7 contains 3 maximal directed paths of which one includes a directed ciruit:

$$P_1 = S_1 e_{12} S_2 e_{23} S_3$$

$$P_2 = S_1 e_{12} S_2 e_{22} S_2 e_{23} S_3 \qquad (5.2)$$

$$P_3 = S_1 e_{13} S_3$$

Furthermore, all these paths are pairwise distinct since $E(P_1) = \{e_{12}, e_{23}\}$, $E(P_2) = \{e_{12}, e_{22}, e_{23}\}$, $E(P_3) = \{e_{13}\}$. Henceforth, I shall omit the word "distinct" it being always understood when a path is being referred to.

Conditions by which symmetric pairs may be identified are given by the following:

Theorem 5.1

Let S_i, S_j be a pair of vertices in a flowgraph G_s. Then S_i, S_j form a symmetric pair if the following conditions hold:

(i) All maximal paths that include S_i also include S_j.

(ii) All maximal paths that include S_j also include S_i.

(iii) Any directed circuit that includes S_i also includes S_j.

(iv) Any directed circuit that includes S_j also include S_i.

Proof

Let S_i, S_j be such that (i)-(iv) above are satisfied. Suppose that in executing G_s, S_i is executed. Then exactly one of the paths that include S_i will be traversed, and so by (i), S_j will also be executed. If S_i is not in a directed circuit then neither is S_j, by (iv). Hence S_i and S_j will both execute exactly once. If S_i is in a directed circuit then so is S_j by (iii), so that if the

circuit is traversed $n \geq 1$ times, both S_i and S_j execute n times. Thus S_j is executed if (whenever) S_i is executed.

Similarly, suppose that in executing G_s, S_j is executed. By analogous arguments it can be seen that S_i executes if (whenever) S_j executes. Hence S_i executes iff S_j executes, i.e., S_i, S_j form a symmetric pair. □

Theorem 5.2

If S_i, S_j form a symmetric pair then

(i) All maximal paths that include S_i also include S_j.

(ii) All maximal paths that include S_j also include S_i.

Proof

Let S_i and S_j be a symmetric pair. Then by definition

Execution of S_i implies execution of S_j (I)

Execution of S_j implies execution of S_i (II)

Now if condition (i) is not satisfied then there exists at least one path, say P_q which includes S_i but not S_j. If in executing G_s P_q is traversed, then S_i will execute but not S_j, contradicting (I). Similarly if condition (ii) is not satisfied then there exists at least one path say P_r which includes S_j but not S_i. If in executing G_s P_r is traversed S_j will execute but not S_i, contradicting (II). □

Finally, as a special case, the following corollary is obtained from Theorem 5.1.

Corollary 5.1

Vertices S_i, S_j in a flowgraph form a symmetric pair if S_i is a source vertex and S_j a sink vertex in G_s, and there exists no other source or sink vertices in G_s[(1)].

Proof

If S_i and S_j are the unique source and sink vertices respectively then all maximal paths originate at S_i and terminate at S_j, i.e. all maximal paths in G_s include both S_i and S_j thus satisfying conditions (i) and (ii) of Theorem 5.1. Finally since invalence (S_i) = outvalance $(S_j) = 0$, S_i and S_j are both excluded from any directed circuit in G_s. □

5.3 Conditions for Global Parallelism

As stated earlier, symmetric vertices serve as potential candidates for the identification of globally parallel MO's. Thus the first step in global analysis is the detection of symmetric pairs. The problem of identifying all symmetric pairs in an arbitrary flowgraph

---

(1) A source vertex in a directed graph is a vertex of invalence 0. A sink vertex is a vertex of outvalence 0.

involves establishing all possible maximal paths from the source to all sinks, followed by a search for vertex pairs satisfying the conditions of Theorem 5.1. However, even if such symmetric pairs are identified, this may not in fact, lead to a smaller set of microinstructions (with respect to a local analysis of the flowgraph). This point will be further explained below.

Consider a symmetric pair (S_i, S_j) in a flowgraph G_s. Let

$$P_{ij} = \{P_1, P_2, \ldots, P_k\} \tag{5.3}$$

be the set of all paths from S_i and S_j, and call this set, the <u>path set from S_i to S_j</u>. Define the <u>internal vertex set</u> V_{ij} corresponding to P_{ij} as the set of distinct vertices included in the paths $P \varepsilon P_{ij}$, excluding S_i and S_j.

For example, consider the symmetric pair S_1 and S_6 in the flowgraph of Fig. 5.8. The corresponding path set is then

$$P_{16} = \{P_1, P_2, P_3, P_4\} \tag{5.4}$$

where

$$\begin{aligned} P_1 &= S_1 e_{16} S_6 \\ P_2 &= S_1 e_{15} S_5 e_{56} S_6 \\ P_3 &= S_1 e_{12} S_2 e_{23} S_3 e_{35} S_5 e_{56} S_6 \\ P_4 &= S_1 e_{12} S_2 e_{23} S_3 e_{34} S_4 e_{45} S_5 e_{56} S_6 \end{aligned} \tag{5.5}$$

The internal vertex set corresponding to P_{16} is clearly:

$$V_{16} = \{S_2, S_3, S_4, S_5\} \quad . \tag{5.6}$$

Definition 5.3

Let (S_i, S_j) be a symmetric pair in a flowgraph G_s, and V_{ij} the internal vertex set corresponding to the path set P_{ij}. Then μ_k in S_i and μ_ℓ in S_j are global candidates if the execution of the sequences $\mu_k \, S \, \mu_\ell$ and $\mu_k \, \mu_\ell \, S$ are state equivalent[2] for all $S \varepsilon V_{ij}$.

Theorem 5.3

Let (S_i, S_j) be a symmetric pair, and V_{ij} the internal vertex set corresponding to the path set P_{ij}. Then μ_k in S_i, and μ_ℓ in S_j are global candidates if $\mu_\ell \, \beta \, \mu_p$ for all μ_p in S, for all S in V_{ij}.

Proof

Assume that for all μ_p in S, for all $S \varepsilon V_{ij}$, $\mu_\ell \, \beta \, \mu_p$. Then the execution of μ_ℓ has no effect on the states of the data sources and sinks of any MO appearing in V_{ij}. Hence for all $S \varepsilon V_{ij}$, $\mu_k \, S \, \mu_\ell$ and $\mu_k \, \mu_\ell \, S$ are state equivalent and so μ_k and μ_ℓ are global candidates. □

---

(2) A pair of sequences of MO's say S_1 and S_2 are said to be state equivalent if for all initial machine states they produce the same final machine state.

It should now be clear that from a pragmatic viewpoint, the identification of all possible symmetric pairs in an arbitrary flowgraph may not be justified.

For suppose we establish that a particular vertex pair S_i, S_j are symmetric, and we also determine the corresponding internal vertex set V_{ij}. Let the length of the i-th SLM be ℓ_i. Then if $|V_{ij}| = k$, the number of MO's contained in V_{ij} is $\sum_{i=1}^{k} \ell_i$. To establish whether μ_ℓ in S_j is a global candidate with some μ_k in S_i, μ_ℓ must be compared with each one of the $\sum \ell_i$ MO's in V_{ij}.

It seems reasonable to hypothesize that the probability of μ_ℓ being data independent of all MO's in V_{ij} decreases as $\sum \ell_i$ increases. Hence if $|V_{ij}| = k$ is too large the probability of obtaining a pair of global candidates is likely to be very small. In such a situation, the computational work expended in identifying S_i, S_j as a symmetric pair is most likely to be wasted.

As an example, consider Fig. 5.8. Here (S_1, S_6) are symmetric, as are (S_2, S_5). Since $V_{25} \subset V_{16}$, $|V_{25}| < |V_{16}|$, so the probability of identifying global candidates between (S_2, S_5) is expected to be higher than between (S_1, S_6).

The proposed solution to the above problem is a heuristic one. Symmetric pairs are identified in <u>loop-free microprograms</u> only; furthermore, the identification of global candidates is attempted only between those

symmetric pairs (S_i, S_j) with internal vertex set V_{ij} where $|V_{ij}| \leq 2$. On this basis, (S_1, S_6) in Fig. 5.8 would not be examined for the presence of global candidates while (S_2, S_5) would. Note that by restricting global analysis to loop-free microprograms, symmetric pairs are identifiable on the basis of Theorem 5.1, conditions (i) and (ii) only. The computational complexity of the parallelism-detection procedure is thereby greatly reduced.

The present section is completed with the following definition :

Definition 5.4

Let (S_i, S_j) be a symmetric pair in a flowgraph G_s, and let μ_k in S_i and μ_ℓ in S_j be global candidates. Then μ_k, μ_ℓ are said to be globally parallel, denoted $\mu_k \,||_G\, \mu_\ell$ if for all initial machine states, the execution of a microinstruction $I = \{\mu_k, \mu_\ell\}$ is state-equivalent to the execution of the microinstruction sequence $I_1 = \{\mu_k\}$, $I_2 = \{\mu_\ell\}$.

In other words I am distinguishing between a pair of MO's say μ_k, μ_ℓ, being globally or locally parallel according to whether in the original flowgraph, μ_k, μ_ℓ belonged to separate (symmetric) vertices or to the same vertex respectively.

Clearly, once μ_k, μ_ℓ have been identified as global candidates, the conditions for $\mu_k \,||_G\, \mu_\ell$ are identical to those for local parallelism. That is

$$\mu_k \,||_G\, \mu_\ell \Longleftrightarrow (\mu_k \,\delta\, \mu_\ell) \vee (\mu_k \,\gamma\, \mu_\ell) \ . \tag{5.7}$$

This is because, since μ_k and μ_ℓ are global candidates, μ_ℓ can precede all MO's in the internal vertex set V_{ij}, We can thus construct a "new" SLM $S_i \mu_\ell$ from which (5.7) follows.

5.4 Identification of Symmetric Pairs in Reduced Flowgraphs

Given a directed graph $G=(V,E)$, $V_i, V_j \in V$ are *strongly connected* if and only if there exists a path from V_i to V_j and a path from V_j to V_i in G. A *strongly connected subgraph* $G' = (V',E')$ of G is a subgraph such that all pairs $V_p, V_q \in V'$ are strongly connected. A *strong component* is a *maximal* strongly connected subgraph.

Given a flowgraph, its strong components can be determined by any one of a number of efficient algorithms [50,67]. A *reduced flowgraph* G_s^R is obtained from the original flowgraph G_s by replacing each of its strong components by a single supervertex. The important characteristic of the reduced flowgraph is that it is acyclic. One should also note that the supervertices represent several SLM's.

As an example, consider the flowgraph of Fig. 5.9. Its strong components are represented by the two subgraphs containing vertices $\{S_2,S_3\}$, and $\{S_{13},S_{14},S_{15}\}$ respectively. In the corresponding reduced flowgraph (Fig. 5.10), these components are denoted by (super) vertices S_2' and S_{13}'.

Thus if a given flowgraph contains strong components, it is first transformed to a reduced form. Furthermore if the flowgraph contains $n > 1$ sinks, it is further transformed into a single sink graph by simply adding a dummy vertex with edges from all the n sinks to the dummy vertex. It is assumed that if such a transformation is made, the dummy vertex is suitably identified.

The first step of the procedure identifies all maximal paths in G_s^R. This is done as follows:

Denote the (unique) source and sink vertices in G_s^R by B (for "begin") and E (for "end") respectively. Then a rooted (or directed) tree (call it the maximal path (MP) tree T_s) is constructed such that any path from the root to a terminating vertex (leaf) of T_s identifies a maximal path in G_s^R.

More precisely, an MP tree T_s corresponding to G_s^R is a tree such that

(a) T_s is rooted at a vertex which corresponds to B in G_s^R; call this root B^t.

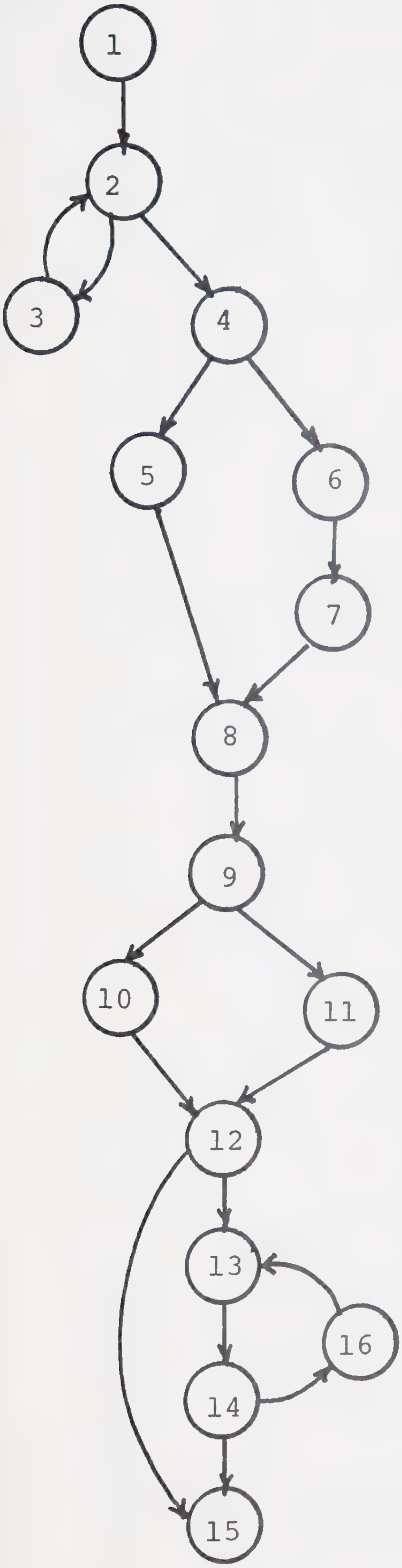

Fig. 5.9

A Flowgraph G_s

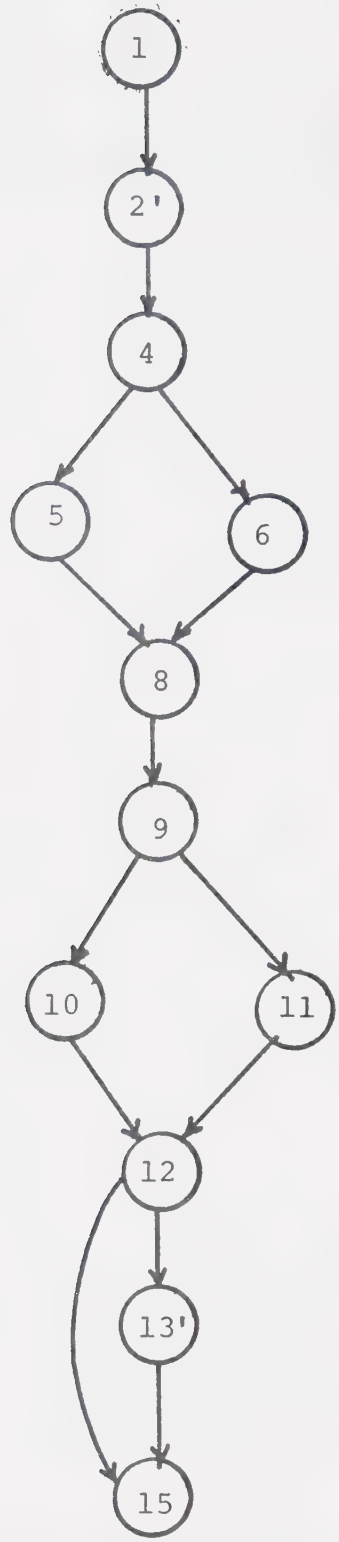

Fig. 5.10

Reduced Flowgraph G_s^R corresponding to G_s

(b) For a vertex S_i^t in T_s there exists an offspring S_j^t in T_s if and only if there is an edge from S_i to S_j in G_s^R.

For convenience of reference, if a vertex in T_s corresponds to a vertex S_i in G_s^R, the former is labelled S_i^t. Note that since T_s is a tree, if there are two vertices S_i, S_j (say) in G_s^R such that edges lead from both S_i and S_j to S_k, then S_k^t appears as a <u>distinct</u> offspring for both S_i^t and S_j^t.

<u>Lemma 5.1</u>

Let T_s be an MP tree corresponding to G_s^R. Then

(a) the leaves of T_s correspond to the sink E of G_s^R.

(b) there exists exactly as many paths from the root to the leaves in T_s as there are maximal paths in G_s^R.

<u>Proof</u>

(a) A leaf, say S_i^t has no offsprings. Hence from the definition of MP tree, S_i in G_s^R is of outvalence 0. Since there is only one sink in G_s^R, every leaf in T_s corresponds to the sink E of $G_s^{R'}$.

(b) Since a tree is connected there is a path in T_s from the root B^t to every leaf in T_s. Let one such path be

$$P_i^t = B^t e_1^t S_1^t e_2^t S_2^t \ldots e_{n-1}^t S_{n-1}^t e_n^t E^t .$$

Then S_1^t is an offspring of B^t, S_2^t is an offspring of $S_1^t, \ldots$, E^t is an offspring of S_{n-1}^t, implying that there exists an edge from B to S_1, from S_1 to $S_2, \ldots$, from S_{n-1} to E in G_s^R, hence the directed path

$$P_i = Be_1S_1e_2S_2 \ldots e_{n-1}S_{n-1}e_nE$$

exists in G_s^R. Since P_i originates at B and terminates at E, it is maximal. Thus if $|P|$ denotes the number of maximal paths in G_s^R, and $|P^t|$ the number of paths in T_s from the root to the leaves, then from the above

$$|P| \geq |P^t| \quad . \tag{5.8}$$

Similarly let

$$P_i = Be_1S_1e_2S_2 \ \ldots \ e_{n-1}S_{n-1}e_nE$$

be a maximal path in G_s^R. Then there exists directed edges from B to S_1, from S_1 to $S_2, \ldots$, from S_{n-1} to E in G_s^R, hence in T_s, E^t is an offspring of $S_{n-1}^t, \ldots$, S_2^t is an offspring of S_1^t, and S_1^t is an offspring of B^t; so a path in T_s exists from B^t to E^t, hence

$$|P^t| \geq |P| \quad . \tag{5.9}$$

From (5.8) and (5.9) it follows that $|P| = |P^t|$. $\square$

The maximal paths can be determined using a modified version of the depth-first search algorithm [3].

Algorithm 5.1

Construction of an MP tree T_s corresponding to a reduced flowgraph G_s^R.

Input

$G_s^R = (V,E)$ represented by adjacency lists ADJ[S] for $S \in V$: vertex $S_k \in$ ADJ[S] iff $(S,S_k) \in E$. T_s is initially empty; furthermore all vertices in the adjacency lists are initially marked "NEW". The operation "SON(θ,ω)" means "create an offspring θ of vertex ω in the tree".

begin

[1] $T_s \leftarrow \{B\}$;

[2] SEARCH(B);

[3] STOP

end

procedure SEARCH(θ)

begin

[4] **for** each NEW vertex $\omega \in$ ADJ[θ] **do**

begin

[5] SON (ω,θ);

[6] Mark ω OLD;

[7] SEARCH (ω)

end

[8] **for** each vertex ω in ADJ[θ] **do** mark ω NEW

end

As an example, consider the reduced flowgraph of Fig. 5.10. The MP tree produced by Algorithm 5.1 is shown in Fig. 5.11.

Verification of this algorithm proceeds by induction on n the number of vertices in the reduced flowgraph. For the purpose of verification assume without loss of generality that vertices are labelled by integers 1,2,...,n, where 1 is the source vertex and n the sink. F further assumption is that the outvalence of all vertices in G_s^R cannot exceed 2. That is, all branches in the original microprogram are two-way branches.

For n = 1, T_s is obviously correctly constructed. Assume as the induction hypothesis that for all reduced flowgraphs with k-1 vertices, Algorithm 5.1 constructs an MP tree rooted at vertex 1. Consider now a k vertex flowgraph. For the subgraph containing vertices {2,3,...,k}, an MP tree is correctly constructed. For the k vertex flowgraph, Step [2] causes SEARCH (1) to be called, and SEARCH is entered. ADJ [1] will contain vertex 2 and possibly, some other vertex i ($3 \le i \le k$). Suppose Step [4] first selects $\omega = 2$. Then Step [5] creates the edge (1,2) in T_s, vertex 2 in ADJ [1] is marked OLD, and SEARCH (2) is called. By the induction hypothesis this call creates an MP tree rooted at vertex 2. Since 1 is connected to 2, on returning from this call, T_s is as shown in Fig. 5.12(a).

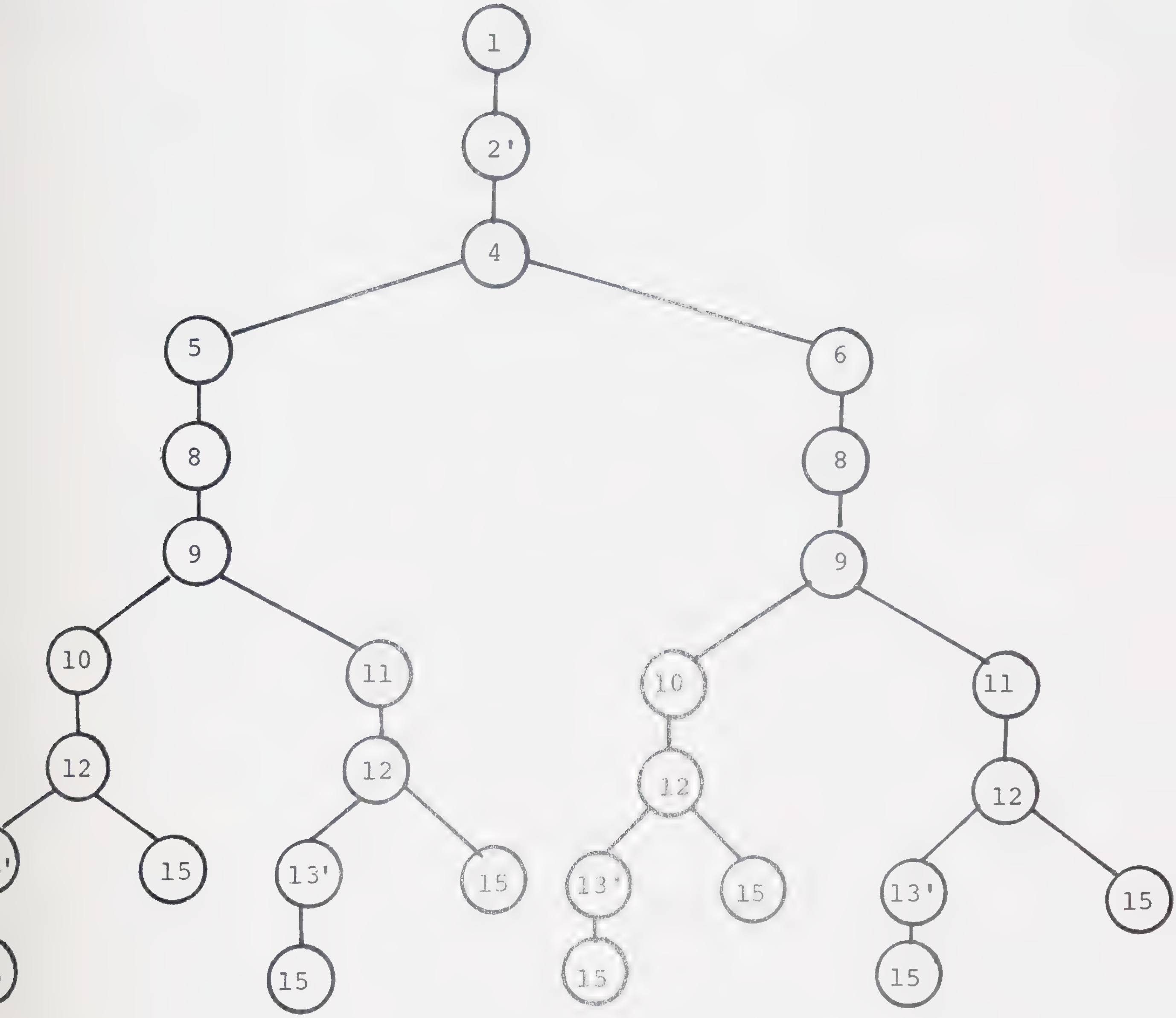

Fig. 5.11

MP Tree T_s for G_s^R of Fig. 5.10

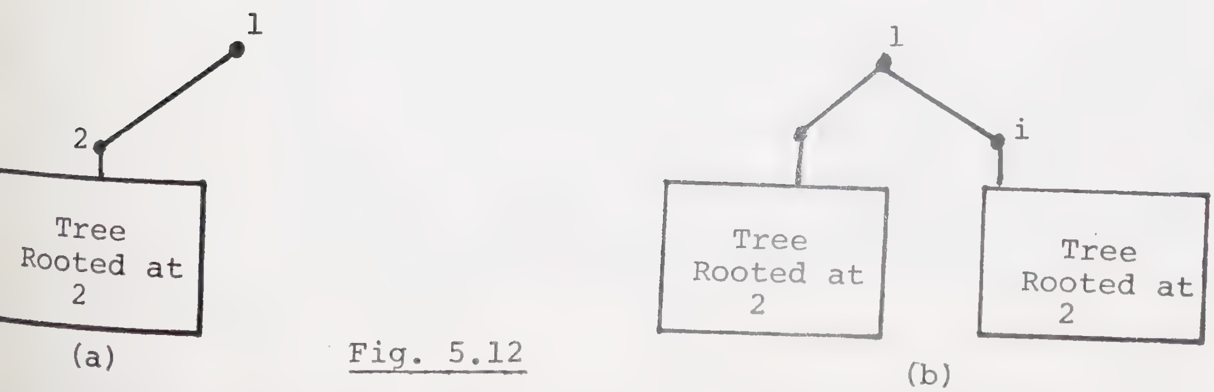

Fig. 5.12

Partial (a) and Final (b) Construction of MP Tree

Step [4] is re-entered; if ADJ [1] contains no other NEW vertex, no other offspring of 1 can exist. Steps [5]-[7] are bypassed, Step [8] is executed, the call SEARCH (1) is completed, and the algorithm terminates. An MP tree rooted at vertex 1 is thus correctly constructed.

If ADJ [1] contains a NEW vertex i, then edge (1,i) is created in T_s (Step [5]), i is marked OLD and SEARCH (i) entered. By the induction hypothesis this call constructs an MP tree rooted at vertex i (Fig. 5.12(b)). On completing SEARCH (i), since ADJ [1] contains no other NEW vertex no further offsprings can exist for 1, hence the tree T_s rooted at 1 is indeed an MP tree. □

Consider the MP tree T_s shown in Fig. 5.11. Each maximal path can be explicitly described by tracing a path from a leaf to the root and reversing the resulting sequence of vertices. From Fig. 5.11 for example, this would yield the following paths (described in terms of the vertices only):

$$
\begin{aligned}
P_1 &= \langle\ 1\ \ 2'\ 4\ \ 5\ \ 8\ \ 9\ \ 10\ \ 12\ \ 15\ \rangle \\
P_2 &= \langle\ 1\ \ 2'\ \ 4\ \ 6\ \ 8\ \ 9\ \ 10\ \ 12\ \ 15\ \rangle \\
P_3 &= \langle\ 1\ \ 2'\ \ 4\ \ 5\ \ 8\ \ 9\ \ 11\ \ 12\ \ 15\ \rangle \\
P_4 &= \langle\ 1\ \ 2'\ \ 4\ \ 6\ \ 8\ \ 9\ \ 11\ \ 12\ \ 15\ \rangle \qquad (5.10) \\
P_5 &= \langle\ 1\ \ 2'\ \ 4\ \ 5\ \ 8\ \ 9\ \ 10\ \ 12\ \ 13'\ \ 15\ \rangle \\
P_6 &= \langle\ 1\ \ 2'\ \ 4\ \ 6\ \ 8\ \ 9\ \ 10\ \ 12\ \ 13'\ \ 15\ \rangle
\end{aligned}
$$

P_7 = < 1 2' 4 5 8 9 11 12 13' 15 >

P_8 = < 1 2' 4 6 8 9 11 12 13' 15 >

However, explicit specification of these paths is not necessary as I shall show below.

Having constructed the MP tree, the next stage is to identify appropriate symmetric pairs and the corresponding internal vertex sets. To do this assume that the k maximal paths (or equivalently, the k leaves in the MP tree) are assigned path numbers 1,2,...,k in any arbitrary manner. A path assigned the number j can then be simply referred to as "path j". Thus, for the above example the path numbers can simply follow the subscripts assigned to the P's in (5.10).

Symmetric pairs can now be easily identified from the MP tree. For, as Theorem 5.1 states, a pair of vertices are symmetric if they are included in exactly the same set of paths. This means that a pair of vertices are symmetric if the vectors of the path numbers that include them are identical. For example, consider the vertex pair <1,2'> in Fig. 5.10. The vectors of their path numbers are, from (5.10), both [1,2,3,4,5,6,7,8] whereas that of vertex 5 is [1,3,5,7]. Thus <1,2'> form a symmetric pair while <1,5> or <2',5> do not.

Instead of actually matching vectors, the symmetric pairs can be more simply identified by assigning a weight of 2^{j-1} to a vertex i if i is included in path j.

Thus if the sum of the weights assigned to vertex i equals the sum of the weights assigned to vertex k, then they are included in precisely the same set of paths, and are therefore symmetric. The weights may be assigned according to the following:

Algorithm 5.2

Assignment of weights to vertices of a reduced flowgraph.

```
For j = 1 step 1 until k do
    begin
        Trace path from leaf to root in path j;
        If path j includes vertex i then WEIGHT [i] ←
                                         WEIGHT [i] + 2^(j-1);
    end
```

The algorithm produces values WEIGHT [1],...., WEIGHT [N] where 1,...,N are the vertices of the original reduced flowgraph G_s^R. For example, the weights assigned to the vertices of Fig. 5.10 are shown in Fig. 5.13. These weights can now be attached to the vertices in G_s^R.

Theorem 5.4

Let weights be assigned to the vertices of the reduced flowgraph according to Algorithm 5.2. Then a pair of vertices have identical weights iff they form a symmetric pair.

PATHS	VERTICES											
	1	2'	4	5	6	8	9	10	11	12	13'	15
1	1	1	1		1	1	1		1	1	1	1
2	2	2	2	2		2	2		2	2	2	2
3	4	4	4		4	4	4	4		4	4	4
4	8	8	8	8		8	8	8		8	8	8
5	16	16	16		16	16	16		16	16		16
6	32	32	32	32		32	32		32	32		32
7	64	64	64		64	64	64	64		64		64
8	128	128	128	128		128	128	128		128		128
WEIGHTS	255	255	255	170	85	255	255	204	51	255	15	255

Fig. 5.13

Computation of Weights

Proof

Let there be k maximal paths in the MP tree and let these be numbered 1,2,...,k in any arbitrary manner. Then the possible weights for a vertex range from 1 to 2^{k-1}, and uniquely identifies the subset of paths that include it. Thus if two vertices S_i, S_j have the same weight they are included in exactly the same set of paths and are therefore symmetric. Conversely, if S_i, S_j are symmetric they must be included in exactly the same set of paths say p,q,...,r and so the weights assigned to both are $2^{p-1} + 2^{q-1} + \ldots + 2^{r-1}$ and are therefore identical. □

Figure 5.14 shows the reduced flowgraph of Fig. 5.10 with the weights indicated in curly brackets. I shall call such a flowgraph, a weighted reduced flow-graph G_S^R.

5.5 Identification of Effective Symmetric Pairs

Within a weighted reduced flowgraph, there may be two or more vertices with identical weights (e.g., the set {1,2',4,8,9,12,15} in Fig. 5.14). In general, let $S = \{S_1, S_2, \ldots, S_n\}$ be a set of such weight equivalent vertices. We may refer to S as a symmetric set since members of S are pairwise symmetric; the problem arises as to how appropriate symmetric pairs may be selected from S.

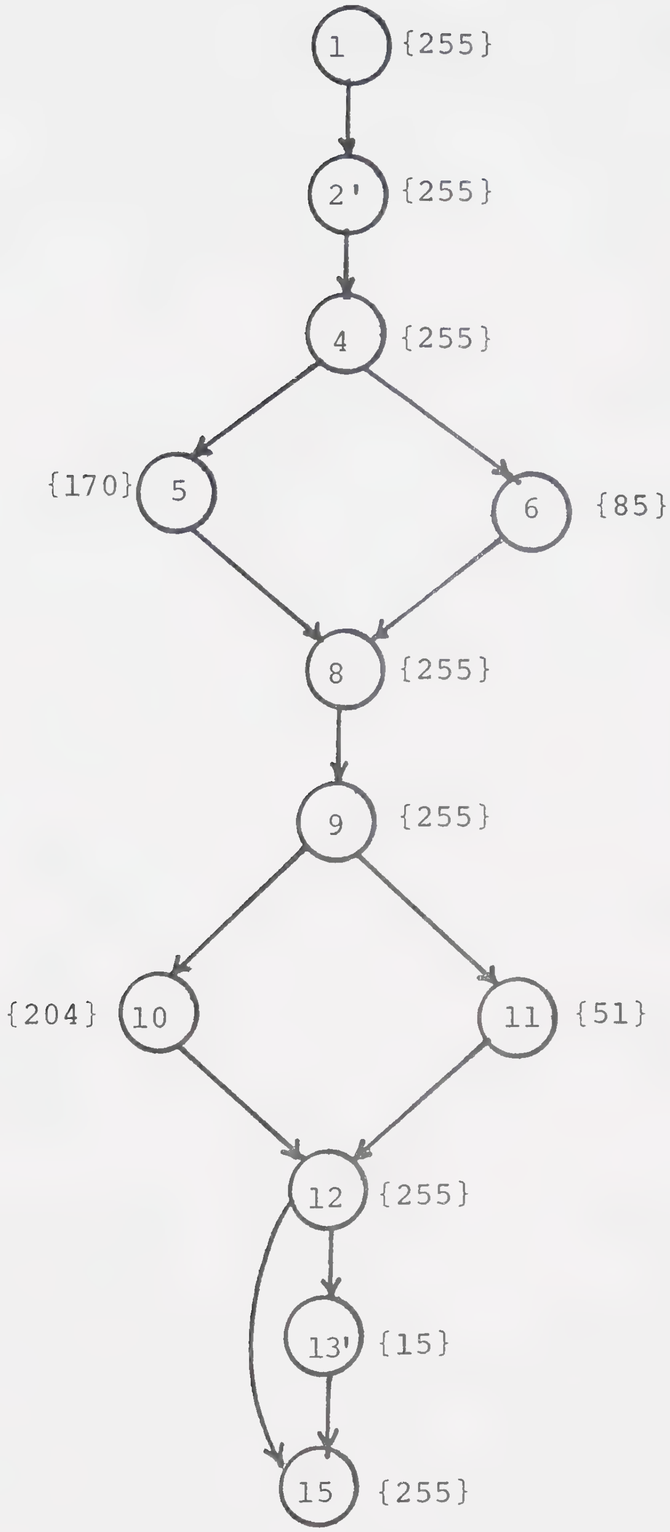

Fig. 5.14

Weighted Reduced Flowgraph, G_S^W

The most obvious - and most expensive - method is to examine systematically, the pairs $<S_1,S_2>$, $<S_1,S_3>$, ...,$<S_{n-1},S_n>$ for parallel MO's. But such a method would be excessively expensive. Instead the following heuristics are used.

[H1] If S_i is a vertex in G_s^W corresponding to a directed circuit or a strong component in G_s, then S_i is ignored in the identification of globally parallel MO's.

[H2] Let $<S_i,S_j>$ be a symmetric pair as determined according to Theorem 5.4, and suppose both S_i and S_j are SLM's. Then if each path from S_i to S_j contains at most one internal vertex, $<S_i,S_j>$ are identified as an effective symmetric pair.

[H3] A vertex S_i can be a member of at most one effective symmetric pair.

[H4] Identification of globally parallel MO's is restricted to effective symmetric pairs.

[H1] is merely a reminder that our analysis is restricted to loop-free microprograms (i.e. acyclic flowgraphs), hence any vertex in the reduced flowgraph that represents a strong component has to be ignored.

[H2] is based upon the discussion presented in Section 5.3 and restricts identification of effective symmetric pairs to those symmetric pairs $\{<S_i,S_j> \mid$ there exists at most one internal vertex in each of the

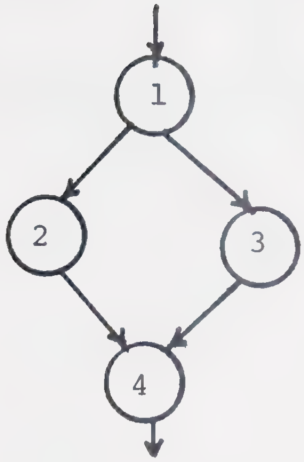

Fig. 5.15

Vertices <1,4> are Effective Symmetric

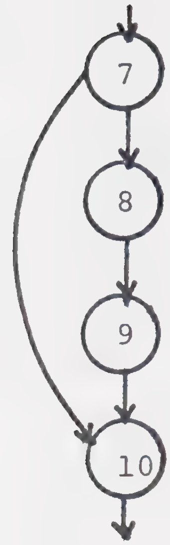

Fig. 5.16

Symmetric Pair <7,10> are Non-effective

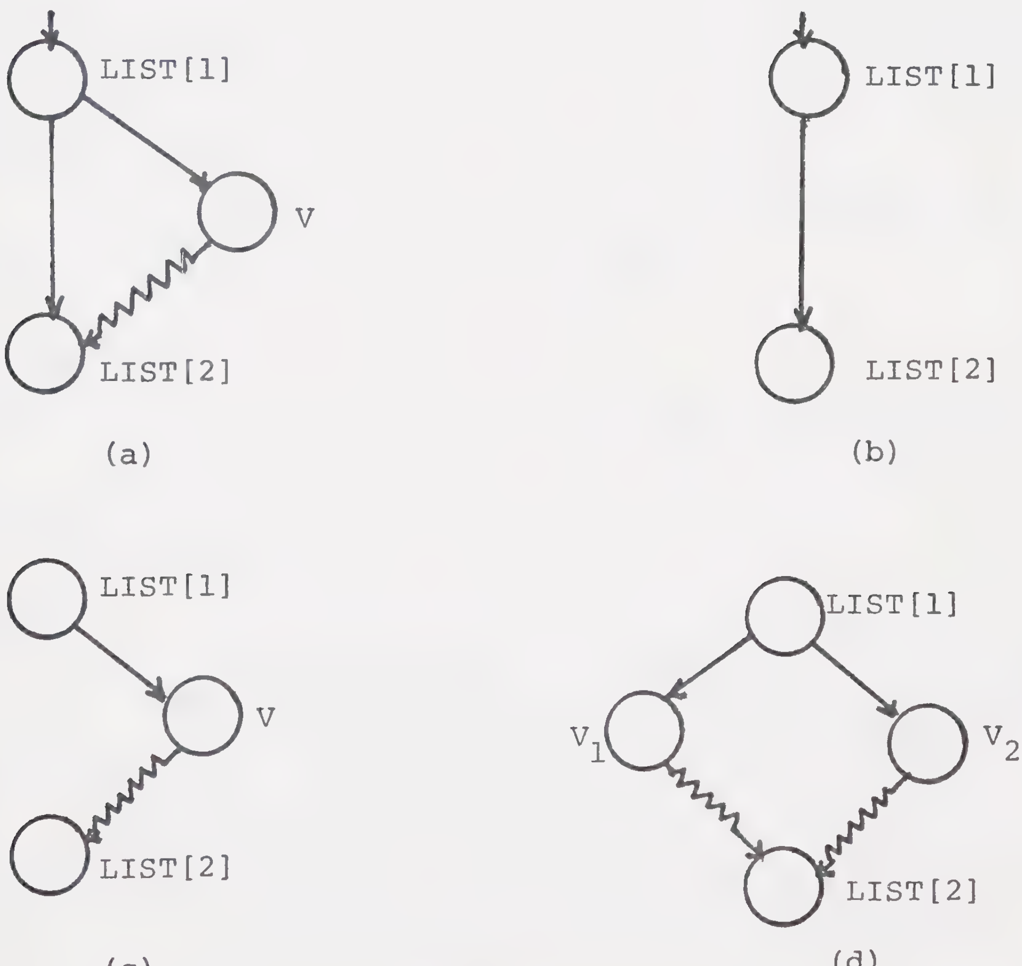

Fig. 5.17

Possible Connections between a Symmetric Pair

paths from S_i to S_j . By this heuristic, vertices <1,4> in Fig. 5.15 are identified as effective symmetric while <7,10> in Fig. 5.16 are not.

Such an identification is not the ad-hoc choice that it seems. For, notice in Fig. 5.16 that <8,9> themselves constitute a symmetric pair, and are furthermore, effective symmetric. In fact if maximal SLM's are identified while constructing the original flowgraph, vertices 8 and 9 would have constituted a single SLM.

[H3] ensures that pairs of effective symmetric vertices are disjoint; finally, identification of globally parallel MO's are restricted by [H4], to effective symmetric pairs only.

The following algorithm identifies effective symmetric pairs in a reduced flowgraph.

<u>Algorithm 5.3</u>

Identification of Effective Symmetric Pairs and their Internal Vertex Sets.

<u>Inputs</u>

(1) The <u>adjacency lists</u> for all vertices in the reduced flowgraph. ADJ [V] is the adjacency list for vertex V.

(2) The <u>sets of symmetric vertices</u> in the reduced flowgraph. Let there be L such sets numbered 1,2,...,L. Each set is ordered such that I[j] refers to the j-th member (vertex) of the I-th set. Furthermore, if the I-th

set contains k_I symmetric vertices, then a symbol distinct from all other symbols denoting vertices, is used as the $(k_I + 1)$-th element of I to indicate the end of the I-th set. By convention, let $I[k_I + 1]$ = "*" for all $1 \leq I \leq L$.

The symmetric sets are easily obtained from the outputs WEIGHT [1],...., WEIGHT [N] of Algorithm 5.2.

The variable LIST is used to access each symmetric set one at a time. INT is used to contain the internal vertex set for an effective symmetric pair. TEMP holds a vertex symbol temporarily. Initially, all elements in ADJ [V] for all vertices V in the reduced flowgraph are marked NEW.

```
[1]     For LIST ← 1 step 1 until L do
          begin
[2]         INT ← ϕ;
[3]         i ← 1; j ← 2;
[4]         If (LIST[i] ≠ *) Λ (LIST[j] ≠ *) then
              begin
[5]             If LIST[i] is a strong component then
                  begin
[6]                 i ← j; j ← j + 1;
[7]                 goto [4]
                  end
```

```
[8]          If LIST [j] is a strong component then
               begin
[9]              j ← j + 1; goto [4]
               end
[10]         If ADJ[LIST[i]] ≠ ϕ then
               begin
[11]             If ∃ some V ε ADJ[LIST[i]] ∋ V is NEW then
                   begin
[12]                 TEMP ← V; Mark V OLD;
[13]                 If TEMP = LIST[j] then goto [11];
[14]                 INT ← INT U {TEMP};
[15]                 If ∃ some W ε ADJ[TEMP] ∋ W = LIST[j]
                       then goto [11]
[16]                 Make all V ε ADJ[LIST[i]] NEW;
[17]                 INT ← ϕ;
[18]                 i ← j; j ← j + 1;
[19]                 goto [4];
                   end
[20]             Output LIST[i], LIST[j], INT;
                 Make all V ε ADJ[LIST[i]] NEW;
               end
[21]             i ← j + 1; j ← i + 1; INT ← ϕ; goto [4]
             end
           end                                              □
```

To verify the correctness of this algorithm, we must show that [H1]-[H3] are satisfied. Furthermore, for each effective symmetric pair identified, the internal vertex set is also identified.

Firstly, note that since the outermost loop (starting from Step [2]) is entered once for each symmetric set, and in the case of the K-th symmetric set no other symmetric set is referenced, all L symmetric sets are examined independently. It is therefore sufficient to consider the K-th symmetric set, i.e., when LIST = K ($1 \leq K \leq L$). The algorithm will be verified by induction on the cardinality of the K-th symmetric set, denoted by $|K|$.

For $|K| = 2$, the two candidate vertices are LIST[1], and LIST[2], and the following possibilities must be considered:

<u>Case I</u>: LIST[1] is a strong component.

<u>Case II</u>: LIST[2] is a strong component but not LIST[1].

For the remaining cases below, neither LIST[1] nor LIST[2] are strong components; V, V_1, V_L are some vertices in the reduced flowgraph other than LIST[1] and LIST[2].

<u>Case III</u>: ADJ[LIST[1]] = {LIST[2],V} (Fig. 5.17(a))

<u>Case IV</u>: ADJ[LIST[1]] = {LIST[2]} (Fig. 5.17(b))

<u>Case V</u>: ADJ[LIST[1]] = {V, LIST[2]} (Fig. 5.17(a))

<u>Case VI</u>: ADJ[LIST[1]] = ϕ

<u>Case VII</u>: ADJ[LIST[1]] = {V} (Fig. 5.17(c))

<u>Case VIII</u>: ADJ[LIST[1]] = $\{V_1, V_2\}$ (Fig. 5.17(d))

<u>Case I</u>: By Steps [5]-[7], the candidate vertices become LIST[2] and LIST[3]. On re-executing Step [4], since LIST[3] = *, the algorithm terminates producing (correctly) no effective symmetric pair. [H1] is thus satisfied.

<u>Case II</u>: Steps [6], [7] are bypassed and Steps [8] and [9] are executed. The candidate vertices become LIST[1] and LIST[3], and again, since LIST[3] = *, the algorithm terminates correctly without producing an effective symmetric pair. Hence [H1] is satisfied.

<u>Case III</u>: Step [11] is entered, and since the condition in this step is satisfied, the block beginning at Step [12] is entered. Step [12] places LIST[2] in TEMP and marks it as OLD in ADJ[LIST[1]]. Since the equality of Step [13] is also satisfied Step [11] is re-entered. At this point, the first path between LIST[1] and LIST[2] satisfies [H2]. On executing Step [11] again, Step [12] is again entered, V placed in TEMP and marked as OLD in ADJ[LIST[1]]. Since TEMP $\neq$ LIST[2], INT = {V} by Step [14], and Step [15] is entered.

Now, if ADJ[V] contains LIST[2], Step [11] is re-entered. At this point ADJ[LIST[1]] contains no NEW vertices, and both paths from LIST[1] to LIST[2] satisfy

[H2]. After executing Step [11], Step [20] is executed and LIST[1], LIST[2] are produced correctly as effective symmetric and INT = {V} as internal vertex. All elements of ADJ[LIST[1]] are marked NEW again; Step [21] results in making INT the empty set, while the new candidate vertices are LIST[3] and LIST[4]. However, on returning to Step [4], since LIST[3] = *, the algorithm (correctly) terminates.

Case IV: As in Case III, LIST[2] is placed in TEMP and marked as OLD in ADJ[LIST[1]]; Step [11] is re-entered. However since no other NEW vertex exists in ADJ[LIST[1]], there is only one path (edge) between LIST[1] and LIST[2], and Step [20] produces as an effective symmetric pair, LIST[1] and LIST[2] and an empty set as the internal vertex set, which is correct. [H2] is thus satisfied.

Case V: Steps [2]-[5], [8], [10]-[11] are executed. By Step [12], TEMP = V, and V is marked OLD in ADJ[LIST[1]]; and by Step [14], INT = {V}.

(a) If ADJ[V] does not contain K[2] then at least one path between LIST[1] and LIST[2] in the reduced flowgraph does not satisfy the condition of [H2]. In the algorithm, Steps [16]-[18] set members of ADJ[K[1]] to NEW, INT to the empty set, and produce as new candidate vertices LIST[2] and LIST[3]. The algorithm terminates correctly without producing an effective symmetric pair.

(b) If on the other hand LIST[2] ε ADJ[V], then this path from LIST[1] to LIST[2] is a valid one. By Step [15], Step [11] is re-entered; Steps [11] and [12] produce TEMP = LIST[2], and by Step [13], Step [11] is entered again. Since all vertices in ADJ[LIST[1]] are now OLD, Step [20] is next entered. At this point, both paths from LIST[1] to LIST[2] satisfy [H2], and INT = {V}. Step [20] thus correctly produces as output LIST[1] and LIST[2] as effective symmetric and {V} as the internal vertex set. ADJ[LIST[1]] is made NEW, and the next pair of vertices become LIST[3] and LIST[4], while INT = ϕ. After executing Step [4], the algorithm terminates.

<u>Case VI</u>: After Step [10] is executed, Step [21] produces as new candidates LIST[3] and LIST[4]. The algorithm terminates producing nothing which is correct.

<u>Case VII</u>: By Step [12], TEMP = V, and V is marked OLD in ADJ[LIST[1]]; by Step [14] INT = {V}.

(a) Now, if LIST[2] $\not\varepsilon$ ADJ[V], then Step [16] is entered after Step [15], and V ε ADJ[LIST[1]] made NEW. The condition of [H2] is of course not satisfied. By Steps [17]-[19], INT = ϕ, and new candidates are LIST[2] and LIST[3]. Since LIST[3] = *, the algorithm terminates correctly producing no output.

(b) If LIST[2] ε ADJ[V] then a valid path is obtained. On returning (by Step [15]) to Step [11] no NEW vertex is found in ADJ[LIST[1]]. Step [20] then produces as

output, <LIST[1], LIST[2]> as the effective symmetric pair with {V} as the internal vertex set, which is correct. Steps [20]-[21] also make $V \varepsilon$ ADJ[LIST[1]] NEW, INT = ϕ, and LIST[3], LIST[4] the new candidate vertices. The algorithm thus terminates correctly.

Case VIII: Step [11] when first entered detects V_1 as NEW; by Step [12] TEMP = V_1 and V_1 in ADJ[LIST[1]] is marked OLD. Since TEMP $\neq$ LIST[2], Step [14] results in INT = $\{V_1\}$.

[a] If LIST[2] ε ADJ[V_1], then one valid path has been found, and after Step [15], Step [11] is re-entered. By Step [12] TEMP = V_2, and V_2 in ADJ[LIST[1]] is marked OLD. Since TEMP $\neq$ LIST[2], Step [14] results in INT = $\{V_1, V_2\}$.

[a(i)] If LIST[2] ε ADJ[V_2], then the second valid path is also found; Step [11] is re-entered. Since no NEW vertices remain, Step [20] is entered and LIST[1],LIST[2] produced as an effective symmetric pair, with $\{V_1, V_2\}$ as the internal vertex set. This is correct.

[a(ii)] If LIST[2] $\not\varepsilon$ ADJ[V_2], then the second path fails to satisfy [H2], and Step [16] follows Step [15]. All elements in ADJ[LIST[1]] is made NEW, INT is made empty, and the new candidates become LIST[2] and LIST[3] by Step [18]. The algorithm thus terminates correctly without producing any output.

[b] If LIST[2] $\not\in$ ADJ[V_1], then the first path is itself not valid, hence no output should be produced. After Step [15], Steps [16]-[19] are executed, resulting in $V_1V_2 \in$ ADJ[LIST[1]] marked NEW, INT made empty, and LIST[2], LIST[3] the new candidate vertices. After Step [4], the algorithm terminates correctly.

This completes the proof for $|K| = 2$. Assume now as the induction hypothesis, that the algorithm is correct for $|K| = n-1$, and consider the case of $|K| = n$.

With the K-th symmetric set as input, i.e. with LIST = K, Algorithm 5.3 would proceed, starting with <LIST[1],LIST[2]> as the initial pair of candidates. Eventually, the first n-1 elements in K will have been processed producing (by the induction hypothesis) the correct (partial) output, and the n-th vertex in LIST will appear as a candidate. If LIST[n] is the first candidate, then the second must be LIST[n+1] = $\star$, and no further output will be produced. Hence the algorithm is correct.

If LIST[n] is the second candidate, then some LIST[i] for $1 \le i \le n-1$ is the first candidate. Moreover LIST[i] cannot be in some effective symmetric pair that has already been identified. For, such a pair is (by the induction hypothesis) produced correctly in Step [20], and the next pair of candidates (by Step [21]), always <u>follow</u> such a pair in the ordered set. Hence, if

<LIST[i], LIST[n]> is a pair of candidates, LIST[i] is not effective symmetric with any LIST[j] ($1 \leq j \leq n-1$). If now <LIST[i],LIST[n]> becomes effective symmetric, then [H3] is guaranteed to be satisfied.

Consider now <LIST[i],LIST[n]> as the candidate pair. Then the possible cases are precisely Case I - Case VIII discussed for $|K| = 2$. By following a similar argument, it is easily seen that <K[i],K[n]> is either identified correctly as an effective symmetric pair, or are both rejected. In either case effective symmetric pairs in LIST are produced satisfying [H1]-[H3]. This completes the proof of correctness of Algorithem 5.3. □

As an example, Algorithm 5.3 may be applied to the weighted reduced flowgraph of Fig. 5.14. The adjacency lists and symmetric sets for this example are shown in Fig. 5.18. The reader may verify that the outputs produced are as follows:

Effective Symmetric Pair	Internal Vertex Set
< 1, 4 >	{2'}
< 8, 9 >	{ }
<12,15 >	{13'}

ADJACENCY LISTS

ADJ [1] = {2'}

ADJ [2'] = {4}

ADJ [4] = {5,6}

ADJ [5] = {8}

ADJ [6] = {8}

ADJ [8] = {9}

ADJ [9] = {10,11}

ADJ [10] = {12}

ADJ [11] = {12}

ADJ [12] = {13',15}

ADJ [13'] = {15}

ADJ [15] = { }

SYMMETRIC SETS

1 : {1,2',4,8,9,12,15,*}

2 : {6,*}

3 : {6,*}

4 : {10,*}

5 : {11,*}

6 : {13,*}

Fig. 5.18

Adjacency Lists and Symmetric Sets for the Weighted Reduced Flowgraph of Fig. 5.14

5.6 The Parallelism-Detection Algorithm

Let $<S_i, S_j>$ be an effective symmetric pair and V_{ij} its internal vertex set as determined by Algorithm 5.3. Consider a pair of MO's μ_k in S_i and μ_ℓ in S_j. To determine whether μ_k and μ_ℓ are globally parallel or not requires determination of:

(a) Whether μ_k and μ_ℓ are global candidates (see Section 5.3, Def. 5.3); that is, by Theorem 5.3 whether for all μ_p in V_{ij}, $\mu_\ell \, \beta \, \mu_p$; and

(b) Whether the condition $(\mu_k \, \delta \, \mu_\ell) \vee (\mu_k \, \gamma \, \mu_\ell)$ is satisfied (see Def. 5.4 and Condition 5.7).

An additional problem is that within the SLM S_j, there may exist some MO $\mu_i < \mu_\ell$ such that μ_i must be executed prior to μ_ℓ, and yet μ_i is not globally parallel to any MO in S_i. Hence a further condition that must be satisfied is:

(c) Whether the movement of μ_ℓ out of S_j into S_i is constrained by one or more MO's within S_j itself.

If there is such a constraint, there is clearly no point in testing for conditions (a) or (b). Similarly, assuming the absence of this constraint, there is no point in testing for condition (b) unless condition (a) holds. We can thus establish a priority ordering for the testing of these three distinct conditions.

A further aspect of the problem is that the total number of microinstructions obtained by global analysis

should be no greater than the number obtained by local analysis only. Suppose the sequence of microinstructions corresponding to S_i (as determined by Algorithm 4.1) is given by

$$I_{si} = (I_{i1}, I_{i2}, \ldots, I_{i,k_i})$$

such that $I_{i1} < I_{i2} < \ldots < I_{i,k_i}$. Then μ_ℓ from S_j can at best be placed in one of these microinstructions, or at worst, in a separate microinstruction, i.e. other than those specified as I_{si}. The latter however, may lead to an <u>increase</u> in the total number of microinstructions. A safer choice in this case is to place μ_ℓ in one of the microinstructions obtained from local optimization of S_j. This at least ensures that the total number of microinstructions will not be larger than that obtained by purely local analysis.

In view of these considerations, the basic approach used by the parallelism-detection algorithm is as follows:

[1] For the sequence of MO's in S_i, use Algorithm 4.1 to obtain a sequence of microinstructions I_{si}.

[2] For each MO μ_ℓ in S_j, invoke Algorithm 4.1 and determine whether μ_ℓ can precede all the microinstructions of S_j obtained thus far - call this set I_{sj}. If not, then continue with Algorithm 4.1 and place μ_ℓ in the earliest possible microinstruction of I_{sj}.

[3] Otherwise, determine whether $\mu_\ell \beta \mu_p$ for all μ_p in V_{ij}. If not, then place μ_ℓ in the earliest microinstruc-

tion of I_{sj}.

[4] Otherwise invoke Algorithm 4.1 in order to place μ_ℓ in some microinstruction in I_{si}. If it cannot be placed in an existing microinstruction, then place it in the earliest microinstruction of I_{sj}.

The complete algorithm is presented below.

Algorithm 5.4

Identification of parallel micro-operations in a symmetric pair $<S_p,S_q>$ with internal vertex set V_{pq}.

Comment

Let $S_q = <\mu_1,\mu_2,\ldots,\mu_t>$; hence t = number of MO's in S_q. Denote the sequence of microinstructions corresponding to S_p and S_q by I_{sp} and I_{sq} respectively. As in Algorithm 4.1, i is a pointer to the microinstruction in I_{sq} "currently" being examined; i' is a pointer to the "latest" microinstruction in I_{sq} at any given time. For I_{sp}, i_2, i_2' serve similar functions except that i_2' will remain invariant since I_{sp} is already determined prior to examining S_q. j points to an element of the input SLM S_q. As in Algorithm 4.1, the expression "branch (x)" denotes a predicate whose value is true if x is a BMO, FALSE otherwise.

[1] Apply Algorithm 4.1 to S_p. Let the resulting sequence of microinstructions be

$$I_{sp} = <I_{p1}, I_{p2},\ldots, I_{p,k_p}>$$

```
[2]   i_2 ← i'_2 ← k_p
[3]   i ← i' ← 1;  j ← 0;  I_1 = φ;
[4]   j ← j + 1;  a ← 0;
      If j > t then I_sq ← {I_1, I_2, ..., I_i'} ; STOP
[5]   If branch (μ_j) then
         begin
            if μ ||_L μ_j ∀ μ ε I_i
                then  I_i ε I_i ∪ {μ_j}
            else
                begin I_{i+1} ← {μ_j}; i' ← i + 1; i ← i' end
            goto [4]
         end
[6]   If (j = 1) V (I_1 = φ) then goto [G1]
[7]   If ∃ μ ε I_i ∋ ∼(μ ||_L μ_j) Λ ∼(μ λ* μ_j)
         then
            begin
                i ← i + 1;  i' ← i;
                I_i ← {μ_j};
                goto [4]
            end
[8]   If (∃ μ ε I_i ∋ μ γ μ_j) Λ (μ' ||_L μ_j ∀ μ' ε I_i - {μ})
         then
            begin
                I_i ← I_i ∪ {μ_j};
                goto [4]
            end
```

```
[9]   If (∃ μ ε I_i ∋ μ δ μ_j Λ SK ∩ SK_j ≠ φ) Λ
            (μ' ||_L μ_j ∀ μ' ε I_i - {μ})
        then
          begin
            I_i ← I_i ∪ {μ_j};
            goto [4]
          end
[10]  While [(μ̇ δ μ_j Λ SK ∩ SK_j = φ) V (μ λ* μ_j) ∀ μ ε I_i] Λ [i > 0]
        do
          begin
            if (μ δ μ_j Λ SK ∩ SK_j =φ) ∀ μ ε I_i then a ←i;
            i ← i - 1
          end
[11]  If i = 0 then
        begin
          if a > 1 then
                  begin
                    I_a ← I_a ∪ {μ_j};
                    i ← i';
                    goto [4]
                  end
          else goto [G1]
        end
```

```
[12]    While ∃ μ ε I_i ∋ ∿(μ||μ_j) do
          begin
            i ← i + 1;
            if i > i' then
              begin     I_i ← {μ_j};
                        i' ← i;
                        goto [4]
              end
          end
[13]    I_i ← I_i ∪ {μ_j};
        i ← i';
        goto [4]
[G1]    If ∃ μ_p ε V_pq ∋ ∿(μ_j β μ_p) then
          begin
            If (j = 1) V (I_1 = φ) then
                                  begin
                                    I_1 ← {μ_j};
                                    goto [4]
                                  end
            else if a ≠ 0 then begin I_a ← I_a ∪ {μ_j}; i ← i';
                                   goto [4]
                               end
```

```
[G1a]        else begin
                   k ← i'
                   while k > 0 do
                      begin
                         I_{k+1} ← I_k;
                         K ← k - 1
                      end
                   I_{k+1} ← {μ_j};
                   i' ← i' + 1;
                   i ← i';
                   goto [4]
               end
         end
[G2]   If ∃ μ ε I_{i_2} ∋ ~(μ ||_L μ_j) Λ ~(μ λ* μ_j) then
         begin
            if j = 1 then
               begin
                  I_1 ← {μ_j};
                  goto [4]
               end
            else goto [G1a]
         end
```

[G3] <u>If</u> $(\exists\, \mu \in I_{i_2} \ni \mu \,\gamma\, \mu_j) \wedge (\mu' \,||_L\, \mu_j \nrightarrow \mu' \in I_{i_2} - \{\mu\})$

<u>then</u>

<u>begin</u>

$I_{i_2} \leftarrow I_{i_2} \cup \{\mu\}_j$:

<u>goto</u> [4]

<u>end</u>

[G4] <u>If</u> $[\exists\, \mu \in I_{i_2} \ni (\mu \,\delta\, \mu_j \wedge SK \cap SK_j \neq \phi)] \wedge [\mu' \,||_L\, \mu_j \nrightarrow$
$\mu' \in I_{i_2} - \{\mu\}]$

<u>then</u>

<u>begin</u>

$I_{i_2} \leftarrow I_{i_2} \cup \{\mu_j\}$

<u>goto</u> [4]

<u>end</u>

[G5] <u>While</u> $[(\mu \,\delta\, \mu_j \wedge SK \cap SK_j = \phi) \vee (\mu \,\lambda^*\, \mu_j) \nrightarrow \mu \in I_{i_2}] \wedge [i_2 > 0]$

<u>do</u>

<u>begin</u>

<u>If</u> $(\mu \,\delta\, \mu_j \wedge SK \cap SK_j = \phi) \nrightarrow \mu \in I_{i_2}$ <u>then</u> $a' \leftarrow i_2$;

$i_2 \leftarrow i_2 - 1$

<u>end</u>

[G6] <u>If</u> $i_2 = 0$ <u>then</u>

<u>begin</u> if $a' \neq 0$ <u>then</u> <u>begin</u>

$I_{a'} \leftarrow I_{a'} \cup \{\mu_j\}$;

$i_2 \leftarrow i_2'$; <u>goto</u> [4]

<u>end</u>

<u>else</u> goto [G1a]

<u>end</u>

[G7] <u>While</u> $\exists\ \mu \in I_{i_2} \ni \sim(\mu || \mu_j)$ <u>do</u>

<u>begin</u>

$i_2 \leftarrow i_2 + 1;$

<u>If</u> $i_2 > i_2'$ <u>then</u> <u>goto</u> [G1a]

<u>end</u>

[G8] $I_{i_2} \leftarrow I_{i_2} \cup \{\mu_j\};$

$i_2 \leftarrow i_2';$

<u>goto</u> [4] □

Verification of this algorithm proceeds along steps similar to those for the verification of Algorithm 4.1 and is therefore omitted here. A few comments are however necessary.

Steps [2]-[13] are almost identical to Algorithm 4.1; these steps construct the microinstruction set I_{sq}. However, if μ_j the MO currently being examined is such that it (a) is the first MO in S_q; or (b) can precede all MO's in the set I_{sq} of microinstructions obtained so far, the algorithm then checks whether $\mu_j \, \beta \, \mu_p$ for μ_p in the internal vertex set V_{pq} (Step [G1]). If this relation holds, the algorithm then proceeds to check whether μ_j can be placed in one of the microinstructions of I_{sp} (Steps [G2]-[G8]). If μ_j is such that it cannot be placed in any of the microinstructions of I_{sp} then substep [G1a] ensures that μ_j is put in an existing microinstruction of I_{sq} if possible or otherwise, in a newly created microinstruction.

5.7 An Example

The reader may obtain a more intuitive idea of the way Algorithm 5.4 works by considering an example. Fig. 5.19 shows the canonical microprogram of Fig. 5.1 represented rather more conventionally. The time validities are indicated in parentheses, while the operational units are implicit.

For this example, clearly:

$$S_p = \langle\mu_1,\mu_2,\mu_3,\mu_4\rangle$$

$$V_{pq} = \langle\mu_5,\mu_6,\mu_7,\mu_8\rangle$$

$$S_q = \langle\mu_9,\mu_{10},\mu_{11},\mu_{12},\mu_{13},\mu_{14}\rangle \quad .$$

Since V_{pq} contains a single vertex which is also an SLM, let us assume (without loss of generality) that Algorithm 4.1 has already been applied to it. Two microinstructions are obtained:

$$I_1'' = \{\mu_5,\mu_6,\mu_7\rangle$$

$$I_2'' = \{\mu_8\} \quad .$$

Thus, on executing Step [1] of Algorithm 5.4, the sequence of microinstructions will be as shown by Fig. 5.21(a). The input string at this point is S_q. The subsequent pattern of construction of the microinstruction sequence is shown as Figs. 5.21 (b)-(f), while the final output is shown as Fig. 5.20.

μ_1 : AIL $\leftarrow$ R1 (V_1);

μ_2 : AO $\leftarrow$ $\underline{\text{Shl}}$ AIL (V_2);

μ_3 : R1 $\leftarrow$ RO (V_1);

μ_4 : $\underline{\text{If}}$ R1 = 0 $\underline{\text{then}}$ $\underline{\text{goto}}$ μ_g (V_1);

μ_5 : AIL $\leftarrow$ R1 (V_1);

μ_6 : AIR $\leftarrow$ R3 (V_1);

μ_7 : R2 $\leftarrow$ AO (V_2);

μ_8 : R2 $\leftarrow$ AO (V_1);

μ_9 : MAR $\leftarrow$ R4 (V_1);

μ_{10} : MBR $\leftarrow$ MEM [MAR] (V);

μ_{11} : MBR $\leftarrow$ R1 (V_1);

μ_{12} : AIL $\leftarrow$ R1 (V_1);

μ_{13} : AIR $\leftarrow$ R2 (V_1);

μ_{14} : AO $\leftarrow$ AIL + AIR (V_2);

Fig. 5.19

An Example of a Canonical Microprogram

$$I_{p_1} = \{\mu_1, \mu_2, \mu_9\}$$

$$I_{p_2} = \{\mu_3, \mu_{10}\}$$

$$I_{p_3} = \{\mu_9\}$$

$$I_1'' = \{\mu_5, \mu_6, \mu_7\}$$

$$I_2'' = \{\mu_8\}$$

$$I_1 = \{\mu_{11}, \mu_{13}\}$$

$$I_2 = \{\mu_{12}, \mu_{14}\}$$

Fig. 5.20

Output from Algorithm 5.4 for the Example of Fig. 5.19

$I_{p_1} = \{\mu_1, \mu_2\}$

$I_{p_2} = \{\mu_3\}$

$I_{p_3} = \{\mu_4\}$

$I_1'' = \{\mu_5, \mu_6, \mu_7\}$

$I_2'' = \{\mu_8\}$

(a)

$I_{p_1} = \{\mu_1, \mu_2, \mu_9\}$

$I_{p_2} = \{\mu_3\}$

$I_{p_3} = \{\mu_4\}$

$I_1'' = \{\mu_5, \mu_6, \mu_7\}$

$I_2'' = \{\mu_8\}$

(b)

$I_{p_1} = \{\mu_1, \mu_2, \mu_9\}$

$I_{p_2} = \{\mu_3, \mu_{10}\}$

$I_{p_3} = \{\mu_4\}$

$I_1'' = \{\mu_5, \mu_6, \mu_7\}$

$I_2'' = \{\mu_8\}$

(c)

$I_{p_1} = \{\mu_1, \mu_2, \mu_9\}$

$I_{p_2} = \{\mu_3, \mu_{10}\}$

$I_{p_3} = \{\mu_4\}$

$I_1'' = \{\mu_5, \mu_6, \mu_7\}$

$I_2'' = \{\mu_8\}$

$I_1 = \{\mu_{11}\}$

(d)

$I_{p_1} = \{\mu_1, \mu_2, \mu_9\}$

$I_{p_2} = \{\mu_3, \mu_{10}\}$

$I_{p_3} = \{\mu_4\}$

$I_1'' = \{\mu_5, \mu_6, \mu_7\}$

$I_2'' = \{\mu_8\}$

$I_1 = \{\mu_{11}\}$

$I_2 = \{\mu_{12}\}$

(e)

$I_{p_1} = \{\mu_1, \mu_2, \mu_9\}$

$I_{p_2} = \{\mu_3, \mu_{10}\}$

$I_{p_3} = \{\mu_4\}$

$I_1'' = \{\mu_5, \mu_6, \mu_7\}$

$I_2'' = \{\mu_8\}$

$I_1 = \{\mu_{11}, \mu_{13}\}$

$I_2 = \{\mu_{12}\}$

(f)

Fig. 5.21

Construction of the Microinstration Sequence for the Example of Fig. 5.19

5.8 Conclusions

The main result of this chapter is the development of a partial theory of global micro-parallelism and its application to the design of algorithms for detecting parallelism in loop-free, canonical microprograms.

The output produced by this system of algorithms may not always be optimal, since several heuristics were used to make the problem analysis more manageable. However in the worst case, the output produced will certainly be at least as good as the output produced by local analysis only.

This last assertion may appear somewhat weak considering the computational work involved in global analysis. But we have already seen an example where better (in fact optimal) output was produced. Moreover, given that parallelism-detection is to be done statically (at compile-time) and that many microprograms will be executed several - probably hundreds of - thousands of times over a machine's operational life time, the overhead incurred in global analysis will probably be justified, where instruction execution efficiency is the main architectural performance objective.

CHAPTER VI

LANGUAGE CONSTRUCTS FOR HORIZONTAL MICROPROGRAMMING

6.1 Introduction

In the last two chapters, I have discussed the design of some algorithms for the identification of parallelism in canonical microprograms. In concluding Chapter V, it was also pointed out that the particular global approach developed here, may not always lead to an optimal output. Moreover, given the computational overheads involved in global analysis, not all microprograms may be suited for such extensive analysis and optimization.

These are practical constraints on the use of mechanical optimization which implementers (of a microprogramming support system) must evaluate, after taking into consideration, the host machine architecture and the nature of the machine level instructions to be implemented.

In this chapter, I will consider a further aspect of micro-parallelism; essentially, this constitutes another addition to the catalogue of techniques for solving the problem of constructing horizontal microprograms. One might call this the linguistic approach to the problem.

To motivate the discussion recall that the execution of a microinstruction in general, may involve both parallelism and sequential activation within a microcycle - particularly if the machine utilizes a polyphase timing scheme (see Section 2.1 and Chapter III). Moreover, in the case of multicycle microinstructions, these same effects may "spread" over several microcycles.

Designers of high-level microprogramming languages have recognized and responded to this fact in - predictably - two different ways. Thus, to avoid explicit denotation of such relationships between micro-operations, Ramamoorthy and Tsuchiya [54] proposed a language by which microcode is specified in canonical form, while the task of extracting horizontal microinstructions was delegated to the translator. Partly influenced by Eckhouse's work on the vertical microprogramming language MPL [26], I had expressed rather similar ideas in an unpublished thesis [22]. This particular approach in fact, provided the impetus for the search for parallelism-detection algorithms, many of which have been described in the earlier chapters.

At the same time, proposals were also made for representing horizontal microinstructions explicitly in the source text [56]. However, Chu's CDL [15] appears to be the only instance of a microprogramming language

containing facilities for specifying the timing characteristics of micro-operations. In CDL, the programmer associates with one or more micro-operations, a "label" designating which part of the microcycle the operations are to be activated in. In other words, CDL allows the expression of polyphase (as well as monophase) horizontal microprograms.

In this chapter, I shall describe a set of constructs which, like CDL, permits horizontal microprograms to be represented explicitly. As stated in Chapter I however, the proposed constructs are motivated by the following considerations:

(1) It seems desirable that a microprogramming language should give the programmer a choice as to whether the horizontal microprogram be specified explicitly or otherwise. Such a choice seems rather important when one realizes that automatic generators of horizontal microcode may not always yield optimal code. Thus, the microprogrammer may wish to optimize critical segments of microcode manually at the source level; in which case, the horizontal microprograms must be specified explicitly.

(2) Given the necessity of devising constructs for horizontal microprogramming, a further crucial characteristic of these constructs must also be considered: for the purpose of microprogram validation and

understanding it is highly desirable that the microprograms be <u>structured</u>. However, because of the complications induced by polyphase timing schemes, structured horizontal microprogramming cannot merely utilize the well known concepts of structured sequential programming [18]. Of far greater relevance are the notions of concurrent programming developed by operating systems theorists [12,13,35].

Thus, a major aim in the design of the proposed constructs is to facilitate the construction of structured horizontal microprograms, "structured" in the sense that for each of the proposed constructs, specific and useful inductive expressions [48] can be defined. As in the case of software design, the ability to make such assertions about the state of the machine should greatly facilitate the verification and understanding of microprograms. This aspect of microprogramming language design has been almost entirely neglected hithertofore.(1)

The following discussion focusses entirely on constructs for expressing the "horizontal" characteristic of microprograms. I shall assume (and this is not a particularly restrictive assumption) that other

---

(1) For a fairly comprehensive review of the status of microprogramming language design, the reader is referred to the very recent monograph "Foundations of Microprogramming" by A.K. Agrawala and T.G. Rauscher (Academic Press, 1976).

constructs exist but that they represent individual, indivisble operations (including branches) whose syntax conforms to say, that of CDL or Ramamoorthy and Tsuchiya's SIMPL language [54].

6.2 A Special Constraint on Construct Formation

Whatever be the form of the constructs that we may choose to propose, they must satisfy the following constraint:

Given a statement (i.e., an instance of a proposed construct), and assuming the existence of an unambiguous mapping of that statement into object microcode, the parallel/serial relationships between components of this microcode must be unambiguously evident in the statement itself.

Such a constraint on the form that constructs may take is imposed from a concern for enhancing both, comprehensiveness, and verifiability, of horizontal microprograms.

In writing a horizontal microprogram, the programmer may conveniently mimic the logic described in Chapters IV and V in obtaining a final product. That is, the programmer may begin with a sequential program; then convert this into an equivalent horizontal program by examining all data dependencies and hardware resource conflicts between the micro-operations. Our present

interest however, lies in the "final" product. For, given a sequence of statements

$$\sigma_1;\ \sigma_2;\ \ldots;\ \sigma_n$$

we must ask (a) whether the individual σ_i's are valid statements; and (b) whether the statement sequence is valid? These questions can be answered if we know the following: given a valid sequence of microinstructions, what conditions must hold between the micro-operations (i) <u>within</u> each microinstruction and (ii) belonging to <u>different</u> microinstructions?

Given a microinstruction

$$I_j = \{\mu_1, \mu_2, \ldots, \mu_n\}$$

the relation $\mu_i \mid\mid_j \mu_k$ is defined for all $\mu_i, \mu_k \in I_j$. Note that the $\mid\mid_j$ relation between some pair of micro-operations μ_i, μ_k holds only in respect to a <u>specific given</u> microinstruction I_j, and merely indicates the fact that μ_i, μ_k have been placed in I_j. If a different micro-instruction I_q contains μ_i but not μ_k, then $\sim(\mu_i \mid\mid_q \mu_k)$.

A microinstruction is said to be <u>valid</u>, if its execution satisfies the following two conditions:

(D1) The state of the machine can be determined exactly after the execution of a known set μ of micro-operations, provided the machine state is known prior to executing μ; and

(D2) No two micro-operations can use an operational unit at the same time.

From earlier discussions on both potential and actual parallelism, it should be obvious that given a valid microinstruction I_j, $\mu_i \,||_j\, \mu_k$ implies

$$[(V_i \cap V_k = \phi)] \vee [(V_i \cap V_k \neq \phi) \wedge (\mu_i \,\beta\, \mu_k) \wedge (U_i \cap U_k = \phi)] \; . \tag{6.1}$$

This follows from the fact that if μ_i, μ_k are in the same microinstruction they must be potentially parallel. (6.1) simply specifies the condition for pairwise potential parallelism.

6.3 Representation of Horizontal Microprograms

Condition (6.1) specifies constraints on the components of a horizontal microinstruction. I shall discuss now, some language constructs that reflect these constraints.

Consider for the present, only those micro-operations which are executable within one microcycle. Following [13], I propose the <u>concurrent microstatement</u>

$$\text{"}\sigma\text{"} \; \underline{\textbf{cobegin}} \; \mu_1; \; \mu_2; \; \ldots; \; \mu_n \; \underline{\textbf{coend}} \tag{6.2}$$

where $\mu_1, \mu_2, \ldots, \mu_n$ are micro-operations, to specify that $\mu_1, \ldots, \mu_n$ are to be executed "concurrently". That is, the time validities of the μ_i's are such that there exists

at least one phase of the microcycle when all the μ_i's will be in execution. The execution of "σ" terminates only when the execution of all the micro-operations in "σ" have terminated.

Given a concurrent microstatement "σ", the condition $V_i \cap V_j \neq \phi$ must hold for all pairs μ_i, μ_j in "σ". Hence a valid concurrent microstatement is one for which the second term of (6.2) holds for all pairs μ_i, μ_j in the statement.

For example, suppose for some particular host machine, the time validities of some of the micro-operations are as shown in Fig. 6.1. Then the statement

$$\underline{\text{cobegin}}\ \ A \leftarrow B;\ F \leftarrow H\ \ \underline{\text{coend}} \tag{6.3}$$

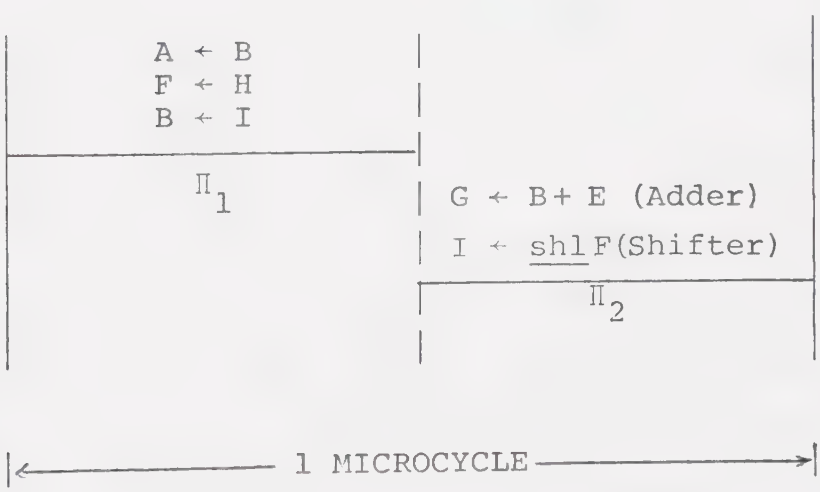

Fig. 6.1

Some Micro-operations and Their Time Validities.

is a valid one (assuming of course that the transfer paths implicit in these two operations are distinct). One the other hand, the statement

$$\underline{\text{cobegin}}\ A \leftarrow B;\ B \leftarrow I;\ I \leftarrow \underline{\text{shl}}\ F\ \underline{\text{coend}} \tag{6.4}$$

is considered invalid since firstly, "$A \leftarrow B$" and "$B \leftarrow I$" are not data-independent though their time-validities are the same; and secondly, even though there are no conflicts between "$B \leftarrow I$" and "$I \leftarrow$ shl F", the fact that their time validities are disjoint, preclude their simultaneous presence in a concurrent microstatement.

Given a concurrent microstatement, we can make rather specific assertions about its effect on the machine state. Prior to illustrating this, let me introduce first, the term microprocess to designate any sequence of events at the register-transfer level, and secondly, the notation (after Hoare [35])

$$\{P\}\ Q\ \{R\} \tag{6.5}$$

which indicates the partial correctness of the microprocess Q with respect to the assertions P and R; i.e., if an assertion P is true of the machine state before start of the microprocess Q, and Q terminates, then the assertion R is true when Q terminates. P and R, are often called the precondition and postcondition respectively of Q, and the entire expression (6.5), an inductive expression.

In the case of the (valid) concurrent microstatement, given that

$$\{P_1\}\ \mu_1\ \{R_1\},\ \{P_2\}\ \mu_2\ \{R_2\},\ldots,\{P_n\}\ \mu_n\ \{R_n\} \qquad (6.6)$$

then

$$\{P_1 \wedge P_2 \wedge \ldots \wedge P_n\}\ \underline{\text{cobegin}}\ \mu_1;\mu_2;\ldots;\mu_n\ \underline{\text{coend}}\ \{R_1 \wedge R_2 \wedge \ldots \wedge R_n\} \qquad (6.7)$$

Furthermore, since $\mu_1,\ldots,\mu_n$ are all executed within a microcycle, the postcondition $R_1 \wedge R_2 \wedge \ldots \wedge R_n$ will be true before the end of that microcycle.

Referring to Fig. 6.1 again, consider the two micro-operations "$A \leftarrow B$" and "$G \leftarrow B + E$". Clearly the time validity of "$A \leftarrow B$" precedes that of "$G \leftarrow B + E$". In a particular situation, a programmer may wish to execute "$A \leftarrow B$" before "$G \leftarrow B + E$" in which case the two operations can be placed in the same microinstruction. However "$A \leftarrow B$" would clearly be executed before "$G \leftarrow B + E$" though both would execute in the same microcycle.

To distinguish between concurrently executable micro-operations, and sequential execution of micro-operations within a microcycle, the latter can be represented by means of the <u>short sequential</u> (SS) microstatement:

$$\underline{\text{shseq}}\ \ \sigma_1\ ;\ \sigma_2\ \underline{\text{end}} \qquad (6.8)$$

where σ_1 is a single micro-operation or a concurrent microstatement; and σ_2 is a single micro-operation, a

concurrent microstatement, or another SS microstatement.

This construct states explicitly that the time validities of all micro-operations in σ_1 precede the time validities of all the micro-operations in σ_2 (i.e., denoted $V(\sigma_1) < V(\sigma_2)$). Furthermore, all micro-operations in $\sigma_1 \cup \sigma_2$ are executed in a microcycle. They must therefore, be placed in one microinstruction.

Consider for example, the following SS microstatement:

```
shseq
  "σ1" cobegin A ← B; D ← E coend;
  "σ2" shseq
         "σ3" cobegin
                   F ← shl [D];
                   E ← B + D;                    (6.9)
                   CAR ← incr CAR.
              coend
         "σ4" MIR ← CM
       end
end
```

This is a valid SS microstatement provided that (a) $V(\sigma_1) < V(\sigma_2)$; (b) $V(\sigma_3) < V(\sigma_4)$ (since σ_2 is itself an SS microstatement); and (c) σ_1 and σ_3 are valid concurrent microstatements.

Assuming that these conditions are satisfied all 6 micro-operations can be placed in the same microinstruction since between any pair of them, condition (6.1) holds.

The inductive expression for the SS microstatement (6.8) is as follows: Since σ_1 and σ_2 are executed in sequence, but are both completed in a microcycle, if $\{P\}\ \sigma_1\ \{Q\}$ and $\{Q\}\ \sigma_2\ \{R\}$ then

$$\{P\}\ \underline{\text{shseq}}\ \ \sigma_1\ ;\ \sigma_2\ \underline{\text{end}}\ \{R\} \tag{6.10}$$

and R is true at the end of the microcycle.

Comparing the two expressions (6.7) and (6.10), it should be evident why a clear distinction between these two situations has been made. For otherwise, if we were to extend the scope of a "valid" concurrent microstatement so as to allow the inclusion of any set of micro-operations such that (6.1) was satisfied then the inductive expression (6.7) would certainly not hold at all times. By providing distinct constructs for these two distinct microprogramming situations, distinct and sharply defined assertions can be made about the machine state. Hence both verifiability and comprehensiveness are greatly enhanced.

It should be pointed out that though I talk about the machine "state", in expressing the pre- and post-conditions, it is sufficient to specify the states of

the sources and sinks of these micro-operations only; the other registers, memories, etc. will of course remain unchanged in content at the time that this particular microstatement is in execution.

Thus far, I have described constructs corresponding to microprocesses that are executed in 1 microcycle. Fig. 6.2 shows a microprocess spanning 3 microcycles: here, a main memory read ("MBR ← MEM[MAR]") operation requires 3 microcycles. However certain other micro-operations are to be executed while the memory read is in progress.

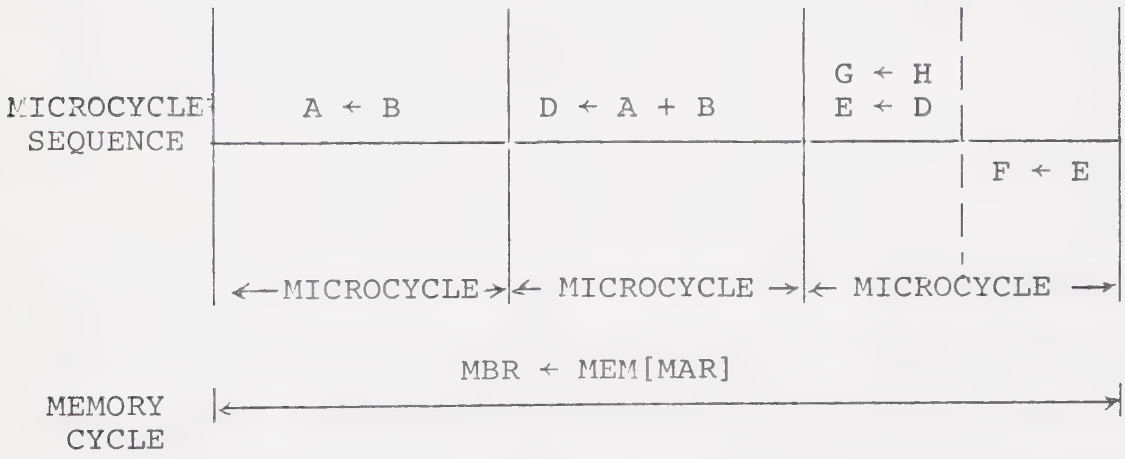

Fig. 6.2

Relationship Between the Microcycle and the Main Memory Cycle

Such concurrency involving multicycle synchronous micro-operations cannot be expressed using the concurrent or SS microstatements since the postconditions of these

statements are true by the end of the microcycle and no later. In fact, note that the postcondition for a concurrent microstatement may not be true _at_ the end of the microcycle though it will have become true earlier in the cycle. This happens for example when σ_1 in (6.8) is a concurrent microstatement. Since $V(\sigma_1) < V(\sigma_2)$, given $\{p\}\ \sigma_1\ \{Q\}$, the assertion Q might not be true at the end of the microcycle.

To express multicycle concurrency, I propose the _extended concurrent (EC) microstatement_:

$$\underline{\text{dur}}\ \sigma_1\ \underline{\text{do}}\ \sigma_2\ \underline{\text{end}} \tag{6.11}$$

where σ_1 is a _micro-operation_, and σ_2 is either another EC microstatment, or a _sequence_

$$\sigma_{21}\ ;\ \sigma_{22}\ ;\ \ldots.;\ \sigma_{2k} \tag{6.12}$$

in which σ_{2i} is either a micro-operation, an SS microstatement, a concurrent microstatement, or the _empty microstatement_ (see below) such that for $1 \le i \le k-1$, the execution of σ_{2i} is completed in a microcycle immediately preceding the microcycle in which $\sigma_{2,i+1}$ is initiated.

Given an EC microstatement, σ_1 and σ_2 will be executed concurrently; execution of the EC microstatement terminates only when _both_ σ_1 and σ_2 have terminated.

In some multicycle programming situations, additional dummy cycles may be necessary to synchronize certain

events. One way of introducing a dummy cycle is through the use of a "NO-OP" microinstruction. The <u>empty microstatement</u> referred to above performs this function: it indicates the execution of an empty set of micro-operations, this "execution" requiring 1 microcycle. It may be denoted simply by the symbol "<u>nil</u>".

The example of Fig. 6.2 can be expressed as:

```
dur MBR ← MEM[MAR]
  do  A ← B;
      D ← A + B;
      shseq
        cobegin                                   (6.13)
            G ← H;
            E ← D
        coend
        F ← E
      end
end
```

The EC microstatement must of course, also satisfy the rule of disjointness; that is, referring to (6.11), if σ_1 designates a particular micro-operation, say μ_n, then for each micro-operation μ_j specified in σ_2, the condition $(V_n \cap V_j \neq \phi) \wedge (\mu_n \beta \mu_j) \wedge (U_n \cap U_j = \phi)$. Because of the disjointness rule, given $\{P_1\}\ \sigma_1\ \{Q_1\}$ and $\{P_2\}\ \sigma_2\ \{Q_2\}$ then

$$\{P_1 \wedge P_2\}\ \underline{\text{dur}}\ \sigma_1\ \underline{\text{do}}\ \sigma_2\ \underline{\text{end}}\ \{Q_1 \wedge Q_2\} \quad . \qquad (6.14)$$

However, since the only timing assertion made about σ_1 is that it will execute concurrent to every micro-operation in σ_2, we cannot make any general statement as to precisely when (relative to the beginning of execution of the EC microstatement) $Q_1 \wedge Q_2$ will hold. For example, σ_1 may continue for a few more cycles after σ_2 has terminated or vice-versa. The most precise general statement that can be made is that the earliest time at which $Q_1 \wedge Q_2$ may possibly be true is when σ_2 has terminated.

When the host machine structure allows only synchronous operations and the duration of σ_1's execution is known, then of course, rather specific timing assertions can be made. Referring to (6.13) for example, if (or since) it is known that a main memory cycle requires 3 microcycles (Fig. 6.2), and since σ_2 requires 3 microcycles to complete, $Q_1 \wedge Q_2$ will be true 3 microcycles after initiating execution of the EC microstatement.

The EC microstatement can also be used to describe parallelism involving asynchronous micro-operations. For example, consider the statement

"σ" dur MBR ← MEM[MAR] do R1 ← R2 end (6.15)

In this case, if "MBR ← MEM[MAR]" happens to be an asynchronous operation, and takes longer than "R1 ← R2", then σ's execution terminates only when the asynchronous

operation terminates.

The programmer can construct entire horizontal microprograms using the above three constructs together with the long sequential (LS) microstatement:

$$\underline{\text{lseq}}\ \sigma_1 \ ;\ \sigma_2\ ;\ \ldots\ ;\ \sigma_n\ \underline{\text{end}} \qquad (6.16)$$

where each σ_i is a micro-operation or one of the microstatements already defined. This statement carries with it, the meaning that for $1 \leq i \leq n-1$:

(a) if neither σ_i nor σ_{i+1} contain asynchronous operations, the components of σ_i complete execution in a microcycle preceding the earliest microcycle in which any component of σ_{i+1} can begin execution;

(b) if σ_i contains an asynchronous component, then σ_{i+1} begins execution only when σ_i terminates.

The LS microstatement is thus the multicycle analogue of the SS microstatement just as the EC microstatement is the multicycle analogue of the concurrent microstatement. Thus, given

$$\{P_1\}\ \sigma_1\ \{P_2\},\ \{P_2\}\ \sigma_2\{P_3\},\ldots,\{P_n\}\ \sigma_n\ \{P_{n+1}\}$$

then

$$\{P_1\}\ \underline{\text{lseq}}\ \sigma_1\ ;\ \sigma_2\ ;\ \ldots\ ;\ \sigma_n\ \underline{\text{end}}\ \{P_{n+1}\}\ . \qquad (6.17)$$

As in the case of the EC microstatement, the precise time (relative to the beginning of the LS statement's

execution) at which P_{n+1} is true cannot be stated in general since one or more of the σ_i's may be EC microstatements. However, if a particular LS microstatement contains no multicycle components, or there are no asynchronous operations and the timing characteristics of the micro-operations are known then time-specific assertions can be made.

One must note the distinction between the SS and LS microstatements. For example, given

$$\underline{\text{shseq}}\ \sigma_1\ ;\ \sigma_2\ \underline{\text{end}}$$

$$\underline{\text{lseq}}\ \sigma_1\ ;\ \sigma_2\ \underline{\text{end}}$$

and assuming that the inductive expressions

$$\{P\}\ \underline{\text{shseq}}\ \sigma_1\ ;\ \sigma_2\ \underline{\text{end}}\ \{R\} \tag{6.18}$$

$$\{P\}\ \underline{\text{lseq}}\ \sigma_1\ ;\ \sigma_2\ \text{end}\ \{R\} \tag{6.19}$$

are both true, then the distinction lies in that in (6.18), R is true at the end of the same microcycle in which σ_1 was initiated while in (6.19) R is not true at the end of σ_1's microcycle, since σ_2 cannot begin execution until at least the following microcycle.

6.4 Representative Examples

Given below are some examples of horizontal microprograms constructed using the statements proposed above. Assertions about the machine state and timing are inserted

at appropriate points in a microprogram, the sequence of these assertions providing a proof of the microprogram's partial correctness.(2)

Examples 1-3 describe microprograms for instruction fetch ("IFETCH"), and the interpretation of (a) a "storage-to-accumulator add" ("ADD") and (b) a "zeroise storage" ("ZEROSTORE") instructions for a simple microprogrammed machine designed originally by Rosin [59] and further elaborated by Flynn [30]. In these three examples however, all micro-operations are assumed to be synchronous. Example 4 repeats IFETCH, assuming this time that main memory read (and write) is asynchronous.

An explanation of the mnemonics used for these examples is given in Fig. 6.3.

SR	:	storage (memory buffer) register
MAR	:	memory address register
AC	:	accumulator
IC	:	instruction counter
IR	:	instruction register
MM	:	main memory
CM	:	control (read only) memory
MIC	:	microinstruction counter (control memory address register)
MIR	:	microinstruction register
REG	:	adder output register

Fig. 6.3

Explanation of Mnemonics for the Rosin/Flynn Machine

(2) Recall that a program is partially correct if it either produces the desired result or fails to terminate. For further discussion of partial and total correctness, the reader is referred to Manna [48] and Owicki and Gries [51].

Example 1: IFETCH (Version 1)

```
        ................  {MIC = a_2; IC = a_1; BEGIN CYCLE 1}
lseq
   MAR ← IC;
            ....................  {MAR = a_1; END CYCLE 1}
   dur SR ← MM[MAR]
     do  INCR IC;
              ..................  {IC = a_1 + 1}
         nil;
         nil;
         nil;
         INCR MIC
         ....................  {MIC = a_2 +1}
   end;
        .....................  {IC = a_1 +1; MIC = a_2 +1;
                                SR = MM[a_1] = i;
                                END CYCLE y ≥ 6}
   IR ← SR;
        .....................  {IR = i}.
   cobegin
     MIC_0-3 ← IR_12-15;
     MIC_4-9 ← 0
   coend;
end
      .....................  {MIC_0-3 = IR_12-15 = OPCODE(i);
                              MIC_4-9 = 0;
                              IC = a_1 + 1; IR = i;
                              END CYCLE y + 2; y ≥ 6}
```

Example 2: ADD

```
.............................{MIC = a2; IR0-11 = a1;
                              ACC = d1; BEGIN CYCLE 1}
lseq
   MAR ← IR0-14
         ....................{MAR = a1; END CYCLE 1}
   dur  SR ← MM[MAR]
     do  nil;
         nil;
         nil;
         nil;
         INCR MIC
           ..............................{MIC = a2 + 1}
   end
     ........................ {SR = MM[a1] = d2;
                               MIC = a2 + 1;
                               END CYCLE y ≥ 6}
   shseq
        cobegin
           ADDLEFT ← SR;
           ADDRT ← AC
        coend
           ................. {ADDLEFT = d2; ADDRT = d1}
        REG ← ADDLEFT + ADDRT
   end
     ........................ {REG = d1+ d2; END CYCLE y+ 1
                                     y ≥ 6}
   AC ← REG;
   goto IFETCH
```

<u>end</u>

$$\cdots\cdots\cdots \left\{\begin{array}{l} AC = d_1 + d_2;\ MIC = a_2 + 1; \\ END\ CYCLE\ y+2,\ y \geq 6; \\ MIC = address\ of\ IFETCH \end{array}\right\}$$

<u>Note</u>: In Flynn's description of Rosin's machine, the overall microprocess "REG ← SR + ACC" is implemented in terms of <u>two</u> micro-operations: "REG ← ADD ← AC" and "REG ← ADD ← SR", executed in the same microcycle, where ADD here, refers to the adder. The implied transfers "ADD ← AC" and "ADD ← SR" are concurrent and take place at the beginning of the microcycle, while the implied transfer "REG ← sum of AC and SR" occur at the end of the cycle.

<u>Example 3: STOREZERO</u>

$$\cdots\cdots\cdots \left\{\begin{array}{l} AC = x;\ IR_{0-11} = a; \\ BEGIN\ CYCLE\ 1 \end{array}\right\}$$

<u>lseq</u>

REG ← ADD ← AC;

.... {REG = x; END CYCLE 1}

AC ← 0;

.... {AC = 0; END CYCLE 2}

<u>cobegin</u>

MAR ← IR_{0-11}

SR ← AC

<u>coend</u>

$$\cdots\cdots\cdots \left\{\begin{array}{l} MAR = a;\ SR = 0; \\ END\ CYCLE\ 3 \end{array}\right\}$$

```
dur  MM[MAR] ← SR
  do ← AC ← REG;
        ........................ {AC = x}
      nil
      nil
      nil
      nil
end;
  ............................ {MM[a] = 0; AC = x;
                                END CYCLE y ≥ 8  }
goto IFETCH
end
  .............................. {MIC = address of IFETCH;
                                  END CYCLE y + 1; y ≥ 8  }
```

Note: The operation REG ← ADD ← AC by itself causes a straightforward transfer of the contents of AC through the adder to REG.

Example 4: IFETCH (Version 2)

```
  ......................... {MIC = a2; IC = a1;}
lseq
    MAR ← IC;
        ..................... {MAR = a1; END CYCLE 1}
    dur  SR ← MM[MAR]
      do  INCR IC end;
          .................. {IC = a1 + 1; SR = MM[a1]
                              = i; END CYCLE y ≥ 2  }
```

INCR MIC;

IR ← SR;

.... $\left\{\begin{array}{l} MIC = a_2 + 1;\ IR = i; \\ END\ CYCLE\ y + 2,\ y \geq 2 \end{array}\right\}$

<u>cobegin</u>

$MIC_{0-3} \leftarrow IR_{12} \leftarrow 15;$

$MIC_{4-9} \leftarrow 0$

<u>coend</u>

<u>end</u>

.... $\left\{\begin{array}{l} MIC_{0-3} = IR_{12-15} = OPCODE(i); \\ MIC_{4-9} = 0; \\ IC = a_1 + 1;\ IR = i; \\ END\ CYCLE\ y + 3,\ y \geq 2 \end{array}\right\}$

<u>NOTE</u>: In this version of IFETCH, notice that the timing assertions are less specific than in Example 1. This is because of the EC microstatement "<u>dur</u> SR ← MM[MAR] <u>do</u> INCR IC <u>end</u>". Clearly all that can be said after encountering this statement is that its execution will require at least 1 cycle. Since this version is based on the assumption that "SR ← MM[MAR]" is asynchronous, this is the most precise statement that can be made, unless we have some further information, e.g., that the asynchronous operation will take more than 3 cycles.

It is important to note however, that the asynchronocity of an operation cannot be <u>inferred</u> from the construct.

Example 5 below, implements a "multiply" instruction (MULT), and is based on the machine structure, timing, and the multiplication algorithm discussed by Husson [36, Section 2.4]. Note the use of a polyphase timing scheme to facilitate the add operation within 1 micro-cycle.

A slight alteration to Husson's notation has been made in construction this example, viz., the BZ ("branch on AOB zero") operation is represented here using the more convenient <u>if</u>..<u>then</u> notation. The mnemonics are explained in Fig. 6.4.

LA	:	Left input to the Adder
RA	:	Right input to the Adder
AOB	:	Adder output register
R1, R2, R3	:	General purpose registers

<u>Fig. 6.4</u>

<u>Explanation of Mnemonics for the Husson's Machine</u>

The final example, Example 6, describes the "RAL8" microprogram for the Hewlett-Packard 2116 microprogrammed computer as specified by Parnas and Siewiorek [52].

Example 5: MULT

```
    .....................  {R1 = x ≥ 0;  R2 = y ≥ 0;  AOB = 0;
                            BEGIN CYCLE 1                     }

lseq
    R3 ← AOB;
        ............................ {R3 = 0;  END CYCLE 1}
    shseq
        LA ← R1;
            ..............................{LA = x}
        shseq
            ADD;
                ..............................{AOB = x}
            shseq  R1 ← AOB;
                      ......................{R1 = x ≥ 0}
                   If AOB = 0 then EXIT
            end
        end
    end
        .......................{R1 = x > 0;  R2 = y;  R3 = 0;
                                END CYCLE 2                  }
    shseq [LOOP]
        ............................ {START OF A CYCLE
                                      R1 > 0          }
        cobegin
            LA ← R2;   RA ← R3
        coend;
```

```
        shseq  ADD;
               R3 ← AOB
        end
     end
        ........................{R3 = y + (x - R1)y;
                                 END OF A CYCLE        }
     shseq
        ........................{BEGIN A CYCLE}
        cobegin
            RA ← R1;
            LA ← "1"
        coend;
        shseq
            SUBTR;
            shseq
                R1 ← AOB;
                   ......................{R1 ≥ 0}
                If AOB ≠ 0 then LOOP
            end
        end
     end [Loop]
   ............................{R1 ≥ 0; END OF A CYCLE}
end
   ......................{R1 = 0; R3 = y + (x-1)y = xy;
                          END CYCLE 2 + 2x             }
```

Example 6: RAL8

```
    ............................ {A = x; START CYCLE 1}
                                 {P = y;              }
lseq
   MB ← 0;
   I ← 0;
   I ← MB <15:10>;
   shseq
      RBVS ← A;
         ................................{RBVS = x}
      shseq
         TBVS ← RBVS × 2;
              ...........................{TBVS = 2x}
         A ← TBVS
      end
   end
      .................................... {END CYCLE 4}
   shseq
      RBVS ← A;
           ............................... {RBVS = 2x}
      shseq
          TBVS ← RBVS × 2;
               ........................... {TBVS = 4x}
          A ← TBVS
      end
      .................................... {A = 4x}
end
      .................................... {END CYCLE 5}
```

```
shseq
    RBVS ← A;
        .............................. {RBVS = 4x}
    shseq  TBVS ← RBVS × 2;
              ........................ {TBVS = 8x}
       A ← TBVS
    end
        .............................. {A = 8x}
end
    .................................. {END CYCLE 6}
shseq
    cobegin
        RBVS ← P; SBVS ← 1
    coend;
    shseq  TBVS ← RBVS + SBVS;
              ........................ {TBVS = y + 1}
       P ← TBVS
    end
        .............................. {P = y + 1}
end
end
    .................................. {A = 8x; P = y+1;
                                        END CYCLE 7     }
```

6.5 Conclusions

In this chapter I have proposed several constructs for expressing structured, horizontal microprograms. Since it is possible to have microinstructions in which micro-operations are sequentially executed, for the sake of validation and understanding, such micro-operations should be distinguished from concurrently executed micro-operations. Both these in turn, have to be distinguished from concurrency effects spanning over several cycles. The constructs discussed above serve to distinguish between these categories of "horizontalness".

By associating certain <u>axioms of execution</u> with these statements, assertions about the state of the machine can be made, and informal proofs of microprogram correctness be constructed. The importance of this facility can hardly be overstated.

One of the key features that distinguish microprogramming from "ordinary" programming is the relevance of timing constraints. Except in the simplest machine structures, a time-independent description of a microprogram is practically valueless. The constructs proposed here not only permit relationships between operations over time to be expressed, they also provide the useful facility of allowing assertions to be made <u>about</u> timing. Such assertions may be used for example, in comparing microprograms for the degree of optimization

achieved.

As a final aspect of this discussion, returning to the problem of microprogram translation, one should note that the LS microstatement indicates explicitly to the translating system that the "source" code is already in horizontal form and so the translator should not spend time in attempting to detect parallelism <u>within</u> this code. As I had mentioned in Section 6.1, the programmer should also have the facility of either partially optimizing a microprogram, or not optimizing it at all. In either case, the <u>simple</u> microstatement

$$\underline{\text{begin}}\ \sigma_1\ ;\ \sigma_2\ ;\ \ldots\ ;\ \sigma_n\ \underline{\text{end}} \qquad (6.20)$$

can be used, where σ_i for $1 \leq i \leq n$ denotes either a micro-operation or one of the microstatements defined above. Given a simple microstatement, the translating system must complete the optimization process using for instance, the algorithms described in this thesis; note that if some σ_j happens to be one of the microstatements described earlier, it will itself have been optimized by the programmer, and can be treated as a single micro-operation in subsequent mechanical optimization.

CHAPTER VII

CONCLUSIONS

7.1 Some Remarks on the Taxonomy of Microprogramming Systems

Within the established classification scheme of microprogrammed control units [58], the focus of attention in the present work has been, the class of horizontal, polyphase systems. Within this class, monophase schemes constitute a limiting subclass. But a microprogramming system exhibits many of the attributes of a complete computer system, and indeed, has often been conceptualized as an "inner" comput [22]. From this viewpoint then, we obtain what is essentially a special kind of parallel processing (inner) computer.

As I have remarked in Chapter I, parallel processing is a rather broad concept and several classification schemes have been proposed as convenient frameworks for categorizing machines [6,37,68]. One well known and widely used taxonomy due to Flynn [27,28], classified computers in terms of the amount of parallelism within the instruction stream and/or the data stream. Note that in this context, an <u>instruction stream</u> is simply a sequence of instructions executed by a processing unit, and a <u>data stream</u> is a sequence of operands that are fed to a processor.

By specifying single or multiple streams of instructions and data, the following classes of systems are obtained:

(1) Single Instruction - Single Data Stream (SISD)

(2) Single Instruction - Multiple Data Stream (SIMD)

(3) Multiple Instruction - Single Data Stream (MISD)

(4) Multiple Instruction - Multiple Data Stream (MIMD)

The question is, within which of these categories does the horizontal polyphase microprogramming system fall into?

Suppose we designate the contents of a chunk of control memory by an array:

$$
\begin{array}{l}
I_1 = [I_{11}, I_{12}, I_{13}, \ldots\ldots, I_{1p}] \\
I_2 = [I_{21}, I_{22}, I_{23}, \ldots\ldots, I_{2p}] \\
\quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
\quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
I_n = [I_{n1}, I_{n2}, I_{n3}, \ldots\ldots, I_{np}] \;.
\end{array}
\qquad (7.1)
$$

Here, each row, I_j represents a microinstruction. I_{jk} denotes the micro-operation specified for execution from the k-th field of I_j.[1]

At the microprogram level, since parallel effects are exhibited between micro-operations and not microinstructions, it is the micro-operation that bears analogy

(1) Note that I_{jk} may be the "null" micro-operation, i.e. the micro-operation that does nothing: a NO-OP.

with the instruction at the program level. Hence the sequence of micro-operations that are executed from any one of the columns of the array (7.1), bears analogy with an "instruction stream"; this sequence of micro-operations is routed to a particular part of the machine data flow which is then appropriately activated (see Section 2.1). What we obtain then, is a multiple instruction stream situation.

Classification of the data stream is however, not so easily obtained. For, at the program level, a multiple data stream is unequivocally exemplified by a sequence of vectors in which the vector elements bear no relation to one another, and corresponding elements of successive vector operands constitute a data stream. This is seen for example, in the case of ILLIAC IV [7] which is an SIMD system. In the MIMD class of systems, instances of multiple data streams are, in addition to vectors, data for concurrent, independent tasks, as in the case of parallel evaluation of arithmetic expressions [53] or concurrent execution of independent processes in a speech recognition system. The latter is one of the main applications envisaged for the Carnegie-Mellon University Multiprocessor [76].

In all these cases, multiplicity of the data, or rather the mutual separateness of the data fed to the separate processing units, is evident. At the micro-

program level however, it is difficult to conceive of the operands to the j-th micro-operation of a micro-instruction as not being closely related to the operands to the k-th micro-operation ($j \neq k$). Rather, the operands for these different micro-operations seem to form a single, meaningful data item. For, the input data to a microprogram (which is interpreting some program-level instruction), are presented by the contents of some words in main memory, the contents of the registers within the data flow, and possibly, the partial contents of some of the control memory words. This entire collection - which is in fact a component of the machine state - constitutes a single data entity that is merely fragmented and distributed to the various parts of the data flow.

The sequence of machine states corresponding to the execution of a sequence of microinstructions is thus the closest analogue we can identify to a data stream, and there is only one such stream corresponding to an "inner" computer. One may conclude therefore, that a horizontal microprogramming system approximates most closely, an MISD machine.

7.2 Plans for Future Work

The principal results of this study can be summarized as follows:

(1) Development of the notion of potential parallelism and its use in constructing polyphase timing schemes and in the minimization of control memory word lengths.

(2) An optimizing algorithm for the detection of parallelism in straight-line microprograms.

(3) Analysis of loop-free canonical microprograms and the construction of a method for identifying parallel micro-operations in such microprograms.

(4) The design of a set of language constructs for representing horizontal microprograms.

As extensions to this work, there are in particular, two rather important and promising areas for study:

(A) Implementation of the proposed parallelism-detection algorithms with respect to commercially available microprogrammable machines. It may be noted in passing that while the wider context within which these algorithms are relevant is the design and implementation of high level microprogramming languages, the algorithms can be implemented as an independent processing system.

One of the problems that the implementer must face is that of representation; more precisely, the algorithms assume that micro-operations are represented in the form of 5-tuples <OP,SC,SK,U,V>. For the particular machine being used for implementation, these distinct micro-operations must therefore be individually identified.

This task is not as formidable as it may seem. For instance, I have recently begun a program of study in which as a first step, micro-operations for the Varian 75 [79] were identified (there are surprisingly, less than 150 of them) and converted into the form of 5-tuples. Using this representation, the Jackson-Dasgupta algorithm has been implemented for the Varian system.

Implementation of these algorithms will certainly provide a powerful support feature for microprogramming and emulation. It will also provide a means of experimentation. A particularly interesting range of questions I would like to see answered is: given a machine structure and control memory organization, to what extent will the average degree of actual parallelism (i.e., the average number of micro-operations/microinstruction) be affected by changing from a local, non-optimizing algorithm (e.g., the JD algorithm) to a local optimizing algorithm (Alg. 4.1) and then to the global method (Alg. 5.3)? Will the average degree of parallelism be bounded within rather narrow limits or will there be significant differences? How does the machine instruction type influence the degree of parallelism? And finally, what will be the overheads incurred in global optimization?

As far as I know, the only published work where micro-parallelism has been investigated empirically, is the report by Barr et al [8] who found that, for the particular machine under study - the Argonne Microprocessor

(AMP) - the average degrees of parallelism for problem-oriented microcode (for a graphics application system) and for microcode interpreting a conventional set of machine instructions were not significantly different. This was however, only a static analysis. There is quite evidently, much scope for further study.

(B) The second area relates to the constructs proposed in Chapter VI. These constructs constitute a contribution to the design of microprogramming languages and will in fact, form some of the basic elements in the design of a language currently being planned by this author. In addition, the constructs provide as I have demonstrated, a representational basis for validating microprograms. What seems immediately necessary, is the application of these constructs to microprograms written for some actual machine and explore their adequacy both in respect to representation, and proving microcode correctness.

At the time of writing, microprogramming seems to have reached some sort of a crossroad at which its "significance" is being critically assessed [61]. Rosin's concept of the "reasonable" machine and his contention that microprograms serve to construct a reasonable superstructure on an unreasonable base is well worth considering as a novel and useful notion.

An "unreasonable" machine in this context is one which requires the programmer to have to grapple with

particular gates, buses, race conditions, split cycle memories and other such hardware features - in fact precisely those features that the microprogrammer is presently coping with. Further implications of this concept are that, if reasonable base machines are built (and according to Rosin, they can be), then micro-programming will lose its raison d'etre, hence will not be necessary.

While I find Rosin's concept of the reasonable machine and its realization a useful one, I feel that the fact that microprogramming serves to disguise the unreasonable hardware from the user, is a positive attribute of microprogramming rather than a negative one as he implies. At least, as long as we are unable to build completely reasonable base machines, microprogramming will continue to participate rather significantly in the creation of reasonable virtual machines.

But of course, unless reliable and efficient firmware is guaranteed, large scale use of the technique may simply lead to the layering of one unreasonable machine on top of another. The danger of this has been pointed out quite clearly by Lehman [46]. The work reported in this thesis will I hope, contribute to the catalogue of ideas and techniques that will help in constructing reasonable, efficient, and reliable virtual machines.

REFERENCES

[1] Agrawala, A.K. and Rauscher, T.G., "Microprogramming: Perspective and Status", IEEE Trans. Comput., C-23, 8 (Aug. 1974) 817-837.

[2] Aho, A.V. and Ullman, J.D., The Theory of Parsing, Translation and Compiling, Vol. II: Compiling, Prentice-Hall, Englewood-Cliffs, N.J., 1973.

[3] Aho, A.V., Hopcroft, J.E. and Ullman, J.D., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[4] Allen, F.E., "Program Optimization", Annual Review of Automatic Programming (Vol. 5), Pergamon Press, London, 1969, 239-307.

[5] Allen, F.E. and Cocke, J., "A Catalogue of Optimizing Transformations" in R. Rustin (Ed), Design and Optimization of Compilers, Prentice-Hall, Englewood-Cliffs, N.J., 1-30.

[6] Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing", Computing Surveys, 5, 1 (March 1973), 31-80.

[7] Barnes, G.H., et al, "The ILLIAC IV Computer", IEEE Trans. Comput., C-17, (1968), 746-755.

[8] Barr, R.G., et al, "A Research Oriented Dynamic Microprocessor", IEEE Trans. Comput., C-22, 11 (Nov. 1973), 976-985.

[9] Bashkow, I.R., Sasson, A. and Kronfield, A., "System Design of a FORTRAN Machine, IEEE Trans. Comput., EC-16, 4, (Aug. 1967), 485-499.

[10] Bernstein, A.J., "Analysis of Programs for Parallel Processing", IEEE Trans. Comput., EC-15 (Oct. 1966), 757-762.

[11] Boulaye, G.G., Microprogramming, Macmillan Press, London, 1975.

[12] Brinch Hansen, P., "Concurrent Programming Concepts", Computing Surveys, 5, 4 (Dec. 1973), 223-245.

[13] Brinch Hansen, P., Operating System Principles, Prentice-Hall, Englewood-Cliffs, N.J., 1973.

[14] Broca, F.R. and Merwin, R.E., "Direct Microprogrammed Execution of the Intermediate Text from a High-Level Language Compiler", Proc. ACM SIGPLAN-SIGMICRO Interface Meeting, Harriman, N.Y., May 1973, 145-153.

[15] Chu, Y., Computer Organization and Microprogramming, Prentice-Hall, Englewood-Cliffs, N.J., 1972.

[16] Chu, Y., "Concepts of High Level Language Computer Architecture", SIGMICRO Newsletter, 6, 4 (Dec. 1975), 9-16.

[17] Cocke, J. and Schwartz, J.T., Programming Languages and their Compilers: Preliminary Notes, Courant Institute of Mathematical Sciences, N.Y., 1970.

[18] Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R., Structured Programming, Academic Press, London, 1972.

[19] Das, S.R., "An Approach for Simplifying Switching Functions by Utilizing the Cover Table Representation", IEEE Trans. Comput., C-20, 3 (March 1971), 355-359.

[20] Das, S.R., Banerji, D.K. and Chattopadhyay, A., "On Control Memory Minimization in Microprogrammed Digital Computers", IEEE Trans. Comput., C-22, 9, (Sept. 1973), 845-848.

[21] Dasgupta, S., and Jackson, L.W., "An Algorithm for Identifying Parallel Micro-operations", Tech. Rept. TR73-20, (Dec. 1973), Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada.

[22] Dasgupta, S., "A High Level Microprogramming Language", M. Sc. Thesis, Univ. of Alberta, Feb. 1974.

[23] Dasgupta, S. and Tartar, J., "On the Minimization of Control Memories", Information Processing Letters, 3, 3 (Jan. 1975), 71-74.

[24] Demco, J., "Principles of Multiple Concurrency Emulation", M. Sc. Thesis, University of Alberta, Aug. 1975.

[25] Deo, N., Graph Theory and its Applications to Engineering and Computer Science, Prentice-Hall, Englewood-Cliffs, N.J., 1973.

[26] Eckhouse, R.H., "A High Level Microprogramming Language (MPL)", Proc. AFIPS SJCC, Vol. 36, AFIPS Press, Montvale, N.J., (1971), 169-177.

[27] Flynn, M.J., "Very High Speed Computing Systems", Proc. IEEE, 54 (1966), 1901-1909.

[28] Flynn, M.J., "Some Computer Organizations and their Effectiveness", IEEE Trans. Comput. (Sept. 1972), 948-960.

[29] Flynn, M.J., "Computer Organizations: Trends and Problems", Proc. IFIP Congress 1974, North-Holland, Amsterdam, 1975.

[30] Flynn, M.J., "Interpretation, Microprogramming and the Control of a Computer", Chapter 10 in [65].

[31] Gonzalez, M.J. and Ramamoorthy, C.V., "Program Suitability for Parallel Processing", IEEE Trans. Comput., C-20, 6, (June 1971), 647-654.

[32] Grasselli, A. and Montanari, U., "On the Minimization of Read Only Memories in Microprogrammed Digital Computers", IEEE Trans. Comput., C-22 (1973), 845-848.

[33] Habib, S., "Microprogrammed Enhancements to Higher Level Languages", Proc. 7th Annual Workshop on Microprogramming (Sept.-Oct. 1974), Palo Alto, 80-81.

[34] Hattori, M., Yano, M. and Fujino,K., "MPGS: A High Level Language for Microprogram Generating Systems", Proc. ACM Conf. (Aug. 1972), 572-581.

[35] Hoare, C.A.R., "Towards a Theory of Parallel Programming", in C.A.R. Hoare and R.A. Perrot (Ed) Operating Systems Techniques, Academic Press, N.Y., 1972.

[36] Hoare, C.A.R., "An Axiomatic Basis of Computer Programming", Comm. ACM., 12, 10, (Oct. 1969), 576-580, 583.

[37] Hobbs, L.C. and Theis, D.J., "Survey of Parallel Processor Approaches and Techniques" in L.C. Hobbs (Ed), Parallel Processor Systems, Technologies and Applications, Spartan Books, N.Y., 1970, 3-20.

[38] Husson, S.S., Microprogramming: Principles and Practices, Prentice-Hall, Englewood-Cliffs, N.J., 1970.

[39] Jackson, L.W. and Dasgupta, S., The Identification of Parallel Micro-operations", Information Proc. Letters, 2, 6, (April 1974), 180-184.

[40] Kleir, R.L. and Ramamoorthy, C.V., "Optimization Strategies for Microprograms", IEEE Trans. Comput., C-20, 7,(July 1971), 783-795.

[41] Kohavi, Z., Switching and Finite Automata Theory, McGraw-Hill, N.Y., 1970.

[42] Knuth, D.E., The Art of Computer Programming, Vol.1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1969.

[43] Knuth, D.E., "Structured Programming with goto Statements", Computing Surveys, 6, 4, (Dec. 1974), 261-302.

[44] Kuck, D.J., Muraoka, Y. and Chen, S.C., "On the Number of Operations Simultaneously Executable in Fortran-like Programs and their Resulting Speed-up", IEEE Trans. Comput., C-21, 12 (Dec. 1972).

[45] Langdon, G.G. Jr., Logic Design: A Review of Theory and Practice, Academic Press, N.Y., 1974.

[46] Lehman, M.M., "Microprogramming Trend Considered Dangerous", Comm. ACM, 18, 6 (June 1975), 358-360.

[47] Mallett, P.W., and Lewis, T.G., "Considerations for Implementing a High Level and Microprogramming Language Translation System", Computer, (Aug. 1975), 40-52.

[48] Manna, Z., The Mathematical Theory of Computation, McGraw-Hill, N.Y., 1974.

[49] Marsland, T.A. and Demco, J., "A Contemporary Computer Emulation", Tech. Rept. TR76-01, Dept. of Computing Science, Univ. of Alberta, Edmonton, Canada.

[50] Munro, I., "Some Results in the Study of Algorithms", Tech. Report. No. 32, (Oct. 1971), Dept. of Computer Science, University of Toronto, Toronto.

[51] Owicki, S. and Gries, D.G., "Verifying Properties of Parallel Programs", Comm. ACM, 19, 5, (May 1976), 279-284.

[52] Parnas, D.L. and Siewiorek, D.P., "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems", Comm. ACM, 18, 7, (July 1975), 401-408.

[53] Ramamoorthy, C.V. and Gonzalez, M.J., "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs", Proc. AFIPS FJCC, AFIP Press, Montvale, N.J., 1969, 1-15.

[54] Ramamoorthy, C.V. and Tsuchiya, M., "A High Level Language for Horizontal Microprogramming", IEEE Trans. Comput., C-23, 8, (Aug. 1974), 791-801.

[55] Ramamoorthy, C.V., "A Survey of the Status of Microprogramming", in J.T. Tou (Ed) Advances in Information Systems and Sciences, Vol. 5, Plenum Press, N.Y., 1974.

[56] Rauscher, T.G., and Agrawala, A.K., "On the Syntax and Semantics of Horizontal Microprogramming Languages", Proc. ACM Conf. (Aug. 1973), 52-56.

[57] Riegel, E.W. and Lawson, H.W., "At the Programming Language-Microprogramming Interface", Proc. ACM SIGPLAN-SIGMICRO Interface Meeting, Harriman, N.Y., (May 1973), 2-10.

[58] Redfield, S.R., "A Study in Microprogrammed Processors: A Medium Sized Microprogrammed Processor", IEEE Trans. Comput., C-20, 7, (July 1971), 743-750.

[59] Rosin, R.F., "Contemporary Concepts of Microprogramming and Emulation", Computing Surveys, 1, 4, (Dec. 1969), 197-212.

[60] Rosin, R.F., Frieder, G. and Eckhouse, R.H. Jr., "An Environment for Research in Microprogramming and Emulation", Comm. ACM, 15, 8, (Aug. 1972), 748-760.

[61] Rosin, R.F., "The Significance of Microprogramming", International Computing Symposium 1973, North-Holland, Amsterdam, 1974.

[62] Schwartz, S.J., "An Algorithm for Minimizing Read-Only Memories for Machine Control", IEEE 10th Annual Symp. Switching and Automata Theory (1968), 28-33.

[63] Sitton, W.G., "Strategies for Microprogram Optimization", (Ph. D Thesis), Tech. Rept. TR73-4, Dept. of Computing Science, University of Alberta, Edmonton, April 1973.

[64] Sockut, G.H., "Firmware/Hardware Support for Operating Systems: Principles and Selected History", SIGMICRO-Newsletter, 6, 4 (Dec. 1975), 17-26.

[65] Stone, H.S. (Ed), Introduction to Computer Architecture, SRA Chicago, 1975.

[66] Stone, H.S., "Parallel Computers", Chapter 8 in [65].

[67] Tarjan, R.E., "Depth First Search and Linear Graph Algorithms", SIAM J. Comput., 1, 2, (1972), 146-160.

[68] Thurber, K.J. and Wald, L.D., "Associative and Parallel Processors", Computing Surveys, 7, 4, (Dec. 1975), 215-255.

[69] Tjaden, G.S. and Flynn, M.J., "Detection and Parallel Execution of Independent Instructions", IEEE Trans. Comput., C-19, 10 (Oct. 1970), 889-895.

[70] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal (Jan. 1967), 25-33.

[71] Tsuchiya, M., "Design of a Multilevel Microprogrammable Computer and a High Level Microprogrammable Language", Ph.D Thesis, University of Texas, Austin, Texas, 1972.

[72] Tsuchiya, M. and Gonzales, M.J., "An Approach to the Optimization of Horizontal Microprograms", Proc. 7th Annual Workshop on Microprogramming, Palo Alto (Sept.-Oct. 1974), 98-106.

[73] Weber, H., "A Microprogrammed Implementation of Euler on IBM System/360 Model 30", Comm. ACM, (Sept. 1967), 549-558.

[74] Wilkes, M.V., "The Use of a Writable Control Memory in a Multiprogramming Environment", Proc. 5th Annual Workshop on Microprogramming, Urbana, (Sept. 1972), 62-65.

[75] Wilner, W.J., "Design of the Burroughs B1700", Proc. AFIPS FJCC 1972, Vol. 41, AFIPS Press, Montvale, N.J.

[76] Wulf, W.A. and Bell, C.G., "C. mmp - A Multi-Mini-Processor", Proc. AFIPS FJCC, Vol. 41, 1972, AFIPS Press, Montvale, N.J., 765-777.

[77] Yau, S.S., Schowe, A.C. and Tsuchiya, M., "On Storage Optimization of Horizontal Microprograms", Proc. 7th Annual Workshop on Microprogramming, Palo Alto, (Sept.-Oct. 1974), 98-106.

[78] ————————, QM-1 Hardware Level User's Mannual", Nanodata Corp., Williamsville, N.Y., 1973.

[79] ————————, "Varian Microprogramming Guide", Varian Data Machines, Irvine, Calif., 1975.

[80] ————————, "Microprogramming Handbook, Microdata Corp., Santa Ana, Calif. (2nd Edition), 1971.